Object Pascal 参考手册

（Ver 0.1）

ezdelphi@hotmail.com

Overview（概述）

Using object pascal（使用 object pascal）

Object Pascal 是一种高级编译语言，具有强类型（对数据类型的检查非常严格）特性，支持结构化和面向对象编程。它的优点包括代码的易读性、快速编译，以及支持多个单元文件从而实现模块化编程。

Object Pascal 具有一些特性以支持 Borland 组件框架和 RAD（快速应用程序开发）环境。在很大程度上，本语言参考的说明和示例假定你使用 Borland 公司的开发工具，如 Delphi 和 Kylix。

绝大多数使用 Borland 开发工具的开发者是在 IDE（集成开发环境）环境下编写代码并进行编译。Borland 开发工具帮助我们设置工程和源文件的许多细节，比如维护单元的依赖信息。并且，使用这些工具在程序的组织上还有一些限制，严格说来，这不是 Object Pascal 语言规范的一部分。比如，Borland 开发工具遵循某些文件和程序的命名约定，若你在 IDE 以外编写代码并使用命令行来编译，你可以避开这些限制。

这些帮助主题假设你在 IDE 环境下工作，并且使用 VCL（可视化组件库）和/或 CLX（跨平台组件库）创建应用程序。但有时候，Borland 一些特定的规则和 Object Pascal 的通用规则并不相同。

Program organization（程序组织）

Program organization: Overview（概述）

应用程序通常被分成多个源代码模块，我们称它们为单元（unit）。每个程序以一个程序头（heading）开始，它为程序指定一个名称。在程序头之后是一个可选的 **uses** 子句，然后是一个由声明和命令语句组成的块（block）。**uses** 子句列出了那些链接到程序的单元，这些单元可以被不同的程序共享，并且通常有自己的 **uses** 子句。

uses 子句给编译器提供各模块间的依赖信息，因为这些信息是存于模块自身的，所以，Object Pascal 程序不需要 makefile 文件、头文件以及 include 预处理指令（这些概念你可能要参考 C 语言）。每当一个工程调入 IDE 时，Project Manager 创建一个 makefile，但只有在包含多个工程的工程组中才保存这些文件。

关于程序结构和依赖关系的更多内容，请参考程序和单元。

Pascal source files（Pascal 源文件）

编译器期望在以下三种文件中取得 Pascal 源代码：

- 单元源文件　（文件扩展名为 **.pas**）
- 工程文件　　（文件扩展名为 **.dpr**）
- 包源文件　　（文件扩展名为 **.dpk**）

单元源文件包含程序代码的主要部分，每个程序包含一个工程文件和多个单元文件。工程文件相当

于传统 Pascal 语言的‘主’程序文件，它把各单元文件组织成一个程序。Borland 开发工具自动为每一个应用程序维护一个工程文件。

如果从命令行编译一个程序，你可以把所有源代码放在单元文件（**.pas**）中，但如果用 IDE 创建程序，则必须有一个工程文件（**.dpr**）。

包源文件类似于工程文件，但它们用来创建称为包的特殊的动态链接库（DLL）。

关于包的更多信息，请参考动态链接库和包。

Other files used to build applications（创建程序所需的其它文件）

除了源代码文件，Borland 工具还需要几种非 Pascal 文件来创建程序。它们是自动维护的，包括以下文件：

- 窗体文件，扩展名为 **.dfm**（Delphi）或 **.xfm**（Kylix）
- 资源文件，扩展名为 **.res** （已经编译的资源文件）
- 工程选项文件，扩展名为 **.dof**（Delphi）或 **.kof**（Kylix）

窗体文件或者是一个文本文件，或者是一个编译的资源文件，资源文件能包含位图、字符串等等。每个窗体文件表示一个窗体，通常对应于程序中的一个窗口或对话框。IDE 允许以文本方式察看和编辑窗体文件，并且能以文本或二进制格式保存它。虽然默认是以文本方式保存窗体，但通常不要手动编辑它，更常用的方式是使用 Borland 提供的可视化设计工具。每个工程至少有一个窗体，每个窗体有一个关联的单元文件（**.pas**），默认情况下，单元的文件名和窗体文件名相同。

除了窗体文件，每个工程使用一个资源文件（**.res**）保存位图作为程序的图标。默认情况下，这个文件和工程文件（**.dpr**）同名。要改变程序的图标，可使用 Project Options 对话框。

工程选项文件（**.dof** 或 **.kof**）包含编译器和链接器设置、搜索路径以及版本信息等等。每个工程对应一个选项文件，它和工程文件同名。通常情况下，文件中的选项是通过 Project Options 对话框来完成的。

IDE 中的许多工具保存其它类型的文件。桌面设置文件（**.dsk** 或 **.desk**）包含窗口的排列信息及其它设置项目。桌面设置或者特定于一个工程（和某个工程相关），或者作用于整个环境（environment-wide）（不是特定于某个工程，对所有工程都有效）。这些文件对编译没有影响。

Compiler-generated files（编译器生成的文件）

在第一次生成一个程序或一个标准 DLL 时，编译器为工程中所使用的每个新单元创建一个编译（过的）单元文件 **.dcu**（Windows）或 **.dcu/.dpu**（Linux）。工程中所有的 **.dcu** 文件（Windows）或 **.dcu/.dpu** 文件（Linux）被链接到一个单独的可执行文件或共享库中；当第一次生成一个包时，编译器为包所包含的每个新单元创建一个 **.dcu** 文件（Windows）或 **.dpu** 文件（Linux），然后创建 **.dcp** 文件和包文件（关于库和包的更多信息，请参考库和包）。若使用 **－GD** 开关，链接器生成一个 map 文件和 **.drc** 文件，**.drc** 文件包含字符串资源，能被编译进资源文件中。

当重新生成一个工程（程序、库或者包）时，除非自上次编译后单元文件（**.pas**）发生了改变、或者没发现 **.dcu** 文件（Windows）和 **.dcu/.dpu** 文件（Linux）、或者明确告诉编译器重新编译它，否则，单元文件不会被重新编译。实际上，只要编译器能找到编译（过的）单元文件（**.dcu** 或 **.dpu**），单元源文件不是必需的。

Example programs（实例程序）

About example programs（关于实例程序）

下面的实例演示 Object Pascal 编程的基本特点，它们是一些简单的 Object Pascal 程序，不能在 IDE 下编译，你可以从命令行编译它们。

A simple console application（一个简单的控制台程序）

下面是一个简单的控制台程序，你可以从命令行编译并运行它。

```
program Greeting;

{$APPTYPE CONSOLE}

var MyMessage: string;

begin
  MyMessage := 'Hello world!';
  Writeln(MyMessage);
end.
```

第一行声明程序叫做 *Greeting*；**{$APPTYPE CONSOLE}** 指示字告诉编译器，这是一个控制台程序，它要从命令行运行；接下来的一行声明了一个变量 *MyMessage*，它存储一个字符串（Object Pascal 包含真正的字符串类型）。程序把字符串 "Hello world!" 赋给变量 *MyMessage*，然后使用 *Writeln* 例程把 *MyMessage* 的内容送到标准输出设备（*Writeln* 在 *System* 单元声明，编译器在每个程序中自动包含这个单元）。

你可以把这个程序（源代码）输入到一个叫 Greeting.pas 或 Greeting.dpr 的文件中，然后在控制台输入如下命令编译它：

对于 Delphi：DCC32 Greeting
对于 Kylix：dcc Greeting

执行的结果是输出信息 "Hello world!"。

除了简单，上面这个例子和我们在 Borland 开发工具下写的程序有几个重要的不同：首先，这是一个控制台程序，Borland 开发工具通常创建图形界面的程序，因此，你不大可能调用 *Writeln*（GUI 程序不能调用 Writeln）；而且，整个程序只有一个文件。在一个典型的程序中，程序头，也就是例子中的第一行，将被放在一个单独的工程文件中。工程文件通常不包含实际的程序逻辑，而只是调用在单元文件中定义的方法。

A more complicated example（一个稍微复杂的程序）

下面的实例程序被分成两个文件：一个工程文件，一个单元文件。工程文件可以存为 Greeting.dpr，它看起来这样：

```
program Greeting;

{$APPTYPE CONSOLE}

uses Unit1;

begin
  PrintMessage('Hello World!');
end.
```

第一行声明程序叫做 *Greeting*，它同样是个控制台程序。**uses** Unit1；子句告诉编译器，*Greeting* 包含（引用）一个叫做 *Unit1* 的单元。最后，程序调用 *PrintMessage* 过程，并把字符串 "Hello world!" 传递给它。请注意，*PrintMessage* 过程是从哪里来的？它是在 *Unit1* 单元定义的。下面是 *Unit1* 单元的源代码，你能把它保存在一个叫 Unit1.pas 的文件中：

```
unit Unit1;

interface

procedure PrintMessage(msg: string);

implementation

procedure PrintMessage(msg: string);
begin
  Writeln(msg);
end;

end.
```

Unit1 单元定义一个叫做 *PrintMessage* 的过程，它接收一个字符串作为参数，并把它送到标准输出设备（在 Pascal 中，没有返回值的例程叫过程，有返回值的例程叫函数）。请注意，*PrintMessage* 在 *Unit1* 中声明了两次，第一次是在关键字 **interface** 的下面，这使得它对于引用 *Unit1* 单元的其它模块（比如 *Greeting*）是可用的；第二次声明是在关键字 **implementation** 的下面，它实际定义 *PrintMessage* 过程。

现在你可以在控制台输入如下命令编译 *Greeting*。

对于 Delphi：DCC32 Greeting
对于 Kylix：dcc Greeting

没必要在命令行参数中包含 *Unit1*。当编译器处理 Greeting.dpr 时，它自动查找 *Greeting* 程序所依赖（引用）的单元文件。程序的执行结果和前面的实例相同，它输出信息 "Hello world!"。

A native application（在 IDE 下设计程序）

我们的下一个实例程序是在 IDE 下用 VCL 或 CLX 组件生成的，它使用自动产生的窗体文件和资源文件，因此，你不能仅仅使用源代码来编译它。它阐明了 Object Pascal 的重要特点。除包含多个单元外，这个程序还使用了类和对象。

程序包含一个工程文件和两个单元文件。首先是工程文件：

```
program Greeting;      { 注释写在一对大括号中 }

uses
  Forms,                { 在 Linux 下改为 QForms }
  Unit1 in 'Unit1.pas'  { Form1 所在的单元 },
  Unit2 in 'Unit2.pas'  { Form2 所在的单元 };

{$R *.res}  { 这个指示字链接工程的资源文件 }

begin
  { 对 Application 的调用 }
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TForm2, Form2);
  Application.Run;
end.
```

我们的程序还是叫 *Greeting*。它引用了三个单元：一个是 *Forms* 单元，它是 VCL 和 CLX 的一部分；二是 *Unit1* 单元，它和程序的主窗体（*Form1*）相关联；三是 *Unit2* 单元，它和另一个窗体（*Form2*）相关联。

这个程序调用 *Application* 对象的一系列方法。*Application* 是类 *TApplication* 的一个实例，它在 *Forms* 单元定义（每个工程自动生成一个 *Application* 对象）。这些调用中有两个调用了 *TApplication* 的 *CreateForm* 方法，第一个 *CreateForm* 创建 *Form1*，它是类 *TForm1*（在 *Unit1* 单元定义）的一个实例；第二个 *CreateForm* 创建 *Form2*，它是类 *TForm2*（在 *Unit2* 单元定义）的一个实例。

Unit1 单元看起来像下面的样子：

```
unit Unit1;

interface

uses       { 下面这些单元是 VCL 的一部分 }
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;
{
在 Linux 下，uses 子句看起来像这样：
uses       { 下面这些单元是 CLX 的一部分 }
  SysUtils, Types, Classes, QGraphics, QControls,  QForms, QDialogs;
}

type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  end;

var
```

```
  Form1: TForm1;

implementation

uses Unit2;  { 这是 Form2 定义的地方 }

{$R *.dfm}  { 这个指示字链接 Unit1 的窗体文件 }

procedure TForm1.Button1Click(Sender: TObject);
begin
  Form2.ShowModal;
end;

end.
```

Unit1 单元创建了类 *TForm1*（继承自 *TForm*）和它的一个实例 *Form1*。*TForm1* 包含一个按钮 *Button1*，它是 *TButton* 的一个实例；还包含一个过程 *TForm1.Button1Click*，在运行时，当用户按下 *Button1* 时它将被执行。*TForm1.Button1Click* 隐藏 *Form1* 并显示 *Form2*（调用 *Form2.ShowModal*），*Form2* 在 *Unit2* 单元定义：

```
unit Unit2;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;
{
在 Linux 下，uses 子句看起来像这样：
uses
  SysUtils, Types, Classes, QGraphics, QControls,  QForms, QDialogs;
}

type
  TForm2 = class(TForm)
    Label1: TLabel;
    CancelButton: TButton;
    procedure CancelButtonClick(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  end;

var
  Form2: TForm2;

implementation

uses Unit1;
```

```
{$R *.dfm}

procedure TForm2.CancelButtonClick(Sender: TObject);
begin
  Form2.Close;
end;

end.
```

Unit2 单元创建了类 *TForm2* 和它的一个实例 *Form*2。*TForm2* 包含一个按钮（*CancelButton*，*TButton* 的一个实例）和一个标签（*Label1*，*TLabel* 的一个实例）。*Label1* 将显示 "Hello world!" 标题，但从源代码中你不能看到这一点。标题是在 *Form2* 的窗体文件 Unit2.dfm 中定义的。

Unit2 单元定义了一个过程。在运行时，当用户点击 *CancelButton* 时，*TForm2.CancelButtonClick* 将被调用，它关闭 *Form2*。这个过程（以及 *Unit1* 单元的 *TForm1.Button1Click*）是作为事件句柄，因为它们响应程序运行时发生的事件。事件句柄通过窗体文件（Windows 下是 **.dfm**，Linux 下是 **.xfm**）赋给指定的事件（事件是一种特殊的属性）。

当 *Greeting* 程序启动时，显示 *Form1* 而隐藏 *Form2*（默认情况下，只有工程文件中最先创建的窗体是可见的，它称为主窗口）。当用户点击 *Form1* 上的按钮时，*Form1* 消失而被 *Form2* 取代，后者将显示 "Hello world!" 信息。当用户关闭 *Form2*（点击 *CancelButton* 按钮或窗口标题栏上的关闭按钮）后，*Form1* 重新显示。

Programs and units（程序和单元）

Programs and units: Overview（概述）

一个程序由称为单元的源代码模块构成。每个单元保存在一个单独的文件中并分别进行编译，编译后的单元被链接到程序中。单元允许你

- 把一个大的程序分成多个模块，它们可单独进行编辑
- 创建可以在程序间共享的库
- 不必提供源代码就可以向其它开发者分发共享库

在传统的 Pascal 编程中，所有源代码，包括主程序都存储在 **.pas** 文件中。Borland 工具使用一个工程文件（**.dpr**）来存储‘主’程序，而大部分源代码则保存在单元文件（**.pas**）中。每个程序（或工程）包含一个工程文件和一个或多个单元文件（严格说来，你不必在一个工程中明确引用任何单元，但所有程序自动引用 System 单元）。要生成程序，编译器需要每个单元的源代码（**.pas**）或编译单元（**.dcu** 等）。

Program structure and syntax（程序的结构和语法）

Program structrue and syntax: Overview（概述）

一个程序包括

- 一个程序头（program heading）
- 一个 **uses** 子句（可选），和
- 一个包含声明和命令语句的块（block）

程序头指定程序的名称；**uses** 子句列出了程序引用的单元；块包含声明和命令语句，当程序运行时，这些命令将被执行。IDE 期望在一个工程文件（**.dpr**）中找到以上三种元素。

下面的实例显示了一个叫做 Editor 的程序：

```
1    program Editor;
2
3    uses
4      Forms,                        { 在 Linux 下改成 QForms }
5      REAbout in 'REAbout.pas'      { AboutBox },
6      REMain in 'REMain.pas'        { MainForm };
7
8    {$R *.res}
9
10   begin
11     Application.Title := 'Text Editor';
12     Application.CreateForm(TMainForm, MainForm);
13     Application.Run;
14   end.
```

第 1 行包含程序头；**uses** 子句从第 3 行到第 6 行；第 8 行是编译器指示字，它把工程的资源文件链接到程序中；第 10 行到第 14 行包含命令块，程序运行时将执行它们；最后，和所有源文件一样，工程文件以一个句点（**.**）结束。

实际上，这是一个典型的工程文件。工程文件通常很短，因为绝大部分的程序逻辑位于单元文件中。工程文件是自动产生并自动维护的，很少需要手工编辑。

The program heading（程序头）

程序头指定程序的名称。它以关键字 **program** 开始，后面跟一个有效标志符（指定程序名），并以分号结束。标志符必须和工程文件名相同，在上例中，因为程序叫 Editor，工程文件应该是 EDITOR.dpr。

在标准 Pascal 中，可在程序名的后面包含参数：

```
program Calc(input, output);
```

Borland Object Pascal 编译器忽略这些参数。

The program uses clause（程序的 uses 子句）

uses 子句列出了共同构成程序的单元，这些单元可能包含自己的 **uses** 子句。关于 **uses** 子句，请参考单元引用和 **uses** 子句。

The block（块）

块包含一个简单语句或结构语句，程序运行时将执行它。在大多数程序中，块包含一个复合语句，它（复合语句）由关键字 **begin** 和 **end** 括起来，其中的命令只是简单调用 *Application* 对象的方法（每个工程都有一个 *Application* 变量，它是 *TApplication*、*TWebApplication* 或 *TServiceApplication* 的一个实例）。块也可以包含常量、类型、变量、过程和函数的声明，它们必须位于（块中）命令语句的前面。

Unit structure and syntax（单元的结构和语法）

Unit structure and syntax: Overview（概述）

一个单元由类型（包括类）、常量、变量以及例程（函数和过程）构成，每个单元由它自己的单元文件（**.pas**）定义。

一个单元以单元头（unit heading）开始，后面跟 **interface**、**implementation**、**initialization** 和 **finalization** 部分，后面两个部分是可选的。一个单元的基本结构看起来这样：

```
unit Unit1;

interface
```

```
uses { 这里是单元列表 }

  { 这里是接口部分 }

implementation

uses { 这里是单元列表 }

  { 这里是实现部分 }

initialization
  { 这里是初始化部分 }

finalization
  { 这里是结束化部分 }

end.
```

单元必须以 **end** 后跟一个句点结束（**end.**）。

The unit heading（单元头）

单元头指定单元的名称。它以关键字 **unit** 开始，后面跟一个有效标志符（指定单元名），并以分号结束。使用 Borland 工具创建的程序，标志符必须和单元文件名相同。所以，单元头

unit MainForm;

必须出现在源文件 MAINFORM.pas 中，编译后的单元文件将是 MAINFORM.dcu。

在一个工程中，单元名必须是独一无二的，两个同名的单元不能用在同一个程序中，即使它们的单元文件位于不同的路径下。

The interface section（接口部分）

单元的接口部分从关键字 **interface** 开始，直到实现部分的开头。接口部分声明常量、类型、变量、过程和函数，所有这些对单元的客户（也就是引用此单元的程序或其它单元）是可用的。在接口部分声明的实体被称为'公用'的，因为它们对客户来说，就像自己声明的一样。

在接口部分声明的过程或函数只是一个例程头，它们的代码块（block）在实现部分定义。所以，在接口部分声明过程和函数就像使用 **forward** 指示字，虽然这里它并没有出现。

在接口部分声明一个类时，必须包含它的所有成员。

接口部分可以包含自己的 **uses** 子句，它必须紧跟在关键字 **interface** 之后。关于 **uses** 子句，请参考单元引用和 **uses** 子句。

The implementation section（实现部分）

单元的实现部分从关键字 **implementation** 开始，直到初始化部分的开头；或者，如果没有初始化部分的话，就直到单元的结束。实现部分定义接口部分声明的过程和函数，在这里，你能以任何顺序定义和调用它们。并且，你也可以省略过程和函数的参数列表，但如果包括它们的话，就必须和在接口部分的声明完全相同。

除了定义公用的过程和函数，实现部分可以定义单元的私有内容，包括常量、类型（包括类）、变量、过程和函数，它们对客户（请参考接口部分）是不可见的。

实现部分可以包含自己的 **uses** 子句，它必须紧跟在关键字 **implementation** 之后。关于 **uses** 子句，请参考单元引用和 **uses** 子句。

The initialization section（初始化部分）

初始化部分是可选的。它从关键字 **initialization** 开始，直到结束化部分的开头；或者，如果没有结束化部分的话，就直到单元的结束。初始化部分所包含的命令，将在程序启动时按它们出现的顺序开始执行。举例来说，如果你定义了需要初始化的结构，你可以在初始化部分来完成。

对于一个单元（称为客户）引用的各个单元，它们的初始化将按客户单元中 **uses** 子句引用它们的顺序开始执行。（也就是说，**uses** 子句中列在前面的单元先初始化）

The finalization section（结束化部分）

结束化部分是可选的，并且只有当一个单元具有初始化部分时才能包含它。结束化部分从关键字 **finalization** 开始，直到单元的结束。结束化部分所包含的命令，将在主程序结束时被执行。使用结束化部分来释放在初始化部分分配的资源。

结束化部分的执行顺序和初始化执行的顺序相反。例如，如果你的程序以 A、B、C 的顺序进行初始化，结束化时的顺序则是 C、B、A。

只要初始化部分的代码开始执行，在程序结束时相应的结束化部分就一定要执行。因此，结束化部分必须能够处理没有完全初始化的数据，因为，如果发生运行时错误，初始化部分的代码可能没有完全执行。

Unit references and the uses clause（单元引用和 uses 子句）

Unit references and the uses clause（单元引用和 uses 子句）

uses 子句列出了被程序、库或单元引用的单元（关于库，请参考库和包）。一个 **uses** 子句可以出现在：

- 程序或库的工程文件
- 单元的接口部分，和
- 单元的实现部分

大多数工程文件包含一个 **uses** 子句，大多数单元的接口部分也是如此，单元的实现部分也可以包含自己的 **uses** 子句。

System 单元自动被每个程序所引用，并且不能在 **uses** 子句中明确列出来（System 单元实现文件 I/O、字符串处理、浮点运算、动态内存分配等例程）。其它一些标准单元，比如 *SysUtils*，必须包含在 **uses** 子句中。大多数情况下，当由工程创建和维护源文件时，所有必需的单元将被包含在 **uses** 子句中。

在单元声明以及 **uses** 子句中（尤其是在 Linux 下），单元名称必须和文件名大小写一致。在其它情况（比如使用限定符的标志符），单元名是大小写无关的。要避免在单元引用中出现问题，要明确指出单元文件：

```
uses MyUnit in "myunit.pas";
```

如果像上面这样在工程文件中明确引用一个单元，在其它源文件中就可以像下面那样简单地引用它，而不必考虑大小写问题：

```
uses Myunit;
```

关于 **uses** 子句的内容和使用位置，请参考多重和间接单元引用以及循环单元引用。

The syntax of a uses clause（uses 子句的语法）

一个 **uses** 子句由关键字 **uses**、后面跟一个或多个由逗号隔开的单元名，最后是一个分号构成。举例如下：

```
uses Forms, Main;
uses Windows, Messages, SysUtils, Strings, Classes, Unit2, MyUnit;
uses SysUtils, Types, Classes, QGraphics, QControls, QForms, QDialogs;
```

在程序或库（工程）的 **uses** 子句中，任何单元名后面可以跟关键字 **in** 和一个（单元）源文件名：源文件名用单引号括起来，可包括或不包括路径，路径可以是绝对路径，也可以是相对路径。举例如下：

```
uses Windows, Messages, SysUtils, Strings in 'C:\Classes\Strings.pas', Classes;
uses
  QForms,
  Main,
  Extra in '../extra/extra.pas';
```

当要指定单元源文件时，在单元的名称后面包含 **in**…。因为 IDE 期望单元名和它所在的源文件名相同，所以通常没有理由这样做。只有当单元源文件的位置不明确时，你才需要使用 **in** 关键字。比如，当

- 引用的单元文件和工程文件在不同的目录，并且单元所在的目录没有包含在编译器搜索路径、也不在库搜索路径中
- 编译器的不同搜索路径中有同名的单元
- 你在命令行编译一个控制台程序，并且单元名和它的文件名不同

编译器也根据 **in**…来判断哪些单元是工程的一部分。在工程文件（**.dpr**）的 **uses** 子句中，只有后面包含 **in** 和一个文件名的单元才被认为是工程的一部分，而其它单元只是被工程引用而不属于这个工程。这种区别对编译程序没有影响，但它影响 IDE 工具，比如 Project Manager 和 Project Browser。

在单元的 **uses** 子句中，你不能用 **in** 告诉编译器到哪里寻找单元文件。每个单元文件必须位于编译器搜索路径、或库搜索路径中，或者和引用它的单元文件位于同一路径；而且，单元名必须和它们的单

元源文件同名。

Multiple and indirect unit references（多重和间接单元引用）

在 **uses** 子句中，单元出现的顺序决定了它们的初始化顺序（请参考初始化部分），也影响编译器对标志符的定位。如果两个单元声明了具有相同名称的变量、常量、类型、过程和函数，编译器使用 **uses** 子句中列在后面的那个单元所声明的标志符。若要访问其它单元的标志符，你必须添加限定符：*UnitName.Identifier*。

在 **uses** 子句中，你只需列出被程序或单元直接引用的单元。也就是说，如果单元 A 使用单元 B 中声明的常量、变量、类型、过程或函数，则 A 必须明确引用单元 B；若单元 B 引用单元 C 的标志符，则单元 A 是间接引用单元 C。这种情况下，在单元 A 的 **uses** 子句中不必包含 C，但编译器为了处理 A，它还是必须能找到 B 和 C。

下面的实例演示了间接引用：

```
program Prog;
uses Unit2;
const a = b;
…
unit Unit2;
interface
uses Unit1;
const b = c;
…
unit Unit1;
interface
const c = 1;
…
```

在这个例子中，*Prog* 直接引用单元 *Unit2*，*Unit2* 又直接引用单元 *Unit1*，因此，*Prog* 间接引用 *Unit1*。因为 *Unit1* 没出现在 *Prog* 的 **uses** 子句中，在 *Unit1* 单元声明的标志符对 *Prog* 是不可见的。

要编译一个客户模块，编译器需要定位客户模块所引用的所有单元，不管是直接的还是间接的。但是，除非这些单元的源文件发生了改变，否则，编译器只需要它们的 **.dcu** 文件（Windows）或 **.dcu/.dpu** 文件（Linux），而不是它们的源文件（**.pas**）。

当一个单元的接口部分被修改时，引用它的其它单元必须被重新编译；但若只修改了单元的实现部分或其它部分，引用它的单元没必要重新编译。编译器自动跟踪依赖关系，只有在需要时才重新编译单元。

Circular unit references（循环单元引用）

当单元间直接或间接地互相依赖（或引用）时，我们称这些单元为相互依赖。相互依赖是被允许的，只要在接口部分的 **uses** 子句中不出现循环引用路径。换句话说，从一个单元的接口部分开始，沿着它所依赖的其它单元的接口部分的依赖路径，一定不能重新返回到这个单元。解决相互依赖问题的一种模式就是，每个循环引用必须至少有一个 **uses** 子句出现在实现部分。

在两个单元相互依赖这种最简单的情况下，你不能在它们的接口部分的 **uses** 子句中互相列出对方。所以，下面的例子将产生编译错误：

```
unit Unit1;
interface
uses Unit2;
…
unit Unit2;
interface
uses Unit1;
…
```

但是，若把其中的一个引用移到实现部分，这两个单元之间的相互引用将是合法的：

```
unit Unit1;
interface
uses Unit2;
…
unit Unit2;
interface
…
implementation
uses Unit1;
…
```

为了减少出现循环单元引用的机会，要尽可能在实现部分的 **uses** 子句中引用单元。只有当来自另一个单元的标志符必须在接口部分使用时，才需要在接口部分的 **uses** 子句中引用它。

Syntactic elements（语法元素）

Syntactic elements: Overview（概述）

Object Pascal 使用 ASCII 字符集，包括 A 到 Z、a 到 z、0 到 9、以及其它标准字符，字母是大小写无关的。空格（#32）和控制符（#0 到#31，其中#13 叫回车符或行的结束符）被称为空白符（blank）。

基本语法元素被称为 **token**（如何翻译？），它们组合起来构成表达式、声明和命令语句。命令语句描述算法行为，它是可执行的；表达式是一个语法单元，它出现在命令语句中并表示一个值；声明定义一个标志符（比如函数或变量的名称），可以用在表达式或命令语句中，并在适当的地方为标志符分配内存。

Fundamental syntactic elements（基本语法元素）

Fundamental syntactic elements: Overview（概述）

在最简单层次上，一个程序是一系列由分隔符隔开的 **token** 构成的。在一个程序中，**token** 是有意义的最小文字单元，分隔符可以是空白符，也可以是注释。严格说来，并不是任何时候在两个 **token** 之间都要有一个分隔符。如下面的代码：

```
Size:=20;Price:=10;
```

是完全合法的。但为遵循约定和代码的可读性起见，我们应当如下书写代码：

```
Size := 20;
Price := 10;
```

Token 被分为特殊符号、标志符、关键字（保留字）、指示字、数字、标签和字符串（常量字符串）。只有当 **token** 是常量字符串时，它才可以包含分隔符。紧邻的标志符、保留字、数字和标签，它们之间必须有一个或多个分隔符。

Special symbols（特殊符号）

特殊符号是非文字和数字字符，或这类字符的组合，它们有固定的意义。以下是单字符的特殊符号：

```
# $ & ' ( ) * + , – . / : ; < = > @ [ ] ^ { }
```

以下的组合字符也是特殊符号：

```
(* (. *) .) .. // := <= >= <>
```

上面，‘[’ 相当于 ‘(.’，‘]’ 相当于 ‘.)’；‘(*’ 和 ‘*)’ 分别相当于 ‘{’ 和 ‘}’（表示注释）。

请注意，!（惊叹号）、 ”（双引号）、 %（百分号）、 ?（问号）、 \（反斜杠）、 _（下划线）、 |（通道）和 ~（破折号）不是特殊符号。

Identifiers（标志符）

标志符用来表示常量、变量、字段、类型、属性、过程、函数、程序、单元、库以及包。一个标志符的长度是任意的，但只有前面的 255 个字符是有意义的。标志符必须以字母或下划线（_）开始，后面可以是字母、数字和下划线，但不能包含空格。关键字不能用作标志符。

因为 Object Pascal 是不区分大小写的，所以，象 *CalculateValue* 标志符，它可以是下面的任何形式：

CalculateValue
calculateValue
calculatevalue
CALCULATEVALUE

在 Linux 下，只有作为单元名的标志符要注意大小写。因为单元名和文件名相同，大小写不一致有时会影响编译。（Linux 下的文件名是区分大小写的）

Qualified identifiers（限定符）

当一个标志符（名称相同）在多个地方声明时，使用它时可能要对标志符进行限定。限定标志符的语法为：

*identifier*1.*identifier*2

这里，*identifier*1 限定 *identifier*2。比如，若两个单元分别声明了一个叫做 *CurrentValue* 的变量，你可以通过如下方式指定要使用 *Unit2* 单元的 *CurrentValue*：

Unit2.CurrentValue

限定符可以重复，比如

Form1.Button1.Click

它调用 *Form1* 中 *Button1* 的 *Click* 方法。

如果你没有使用限定符，在块和范围一节中所讲述的范围规则将决定它作如何解释。

Reserved words（关键字，保留字）

下面的关键字不能被重新定义或用作标志符：

and	array	as	asm
begin	case	class	const
constructor	destructor	dispinterface	div
do	downto	else	end
except	exports	file	finalization
finally	for	function	goto
if	implementation	in	inherited
initialization	inline	interface	is
label	library	mod	nil
not	object	of	or

out	packed	procedure	program
property	raise	record	repeat
resourcestring	set	shl	shr
string	then	threadvar	to
try	type	unit	until
uses	var	while	With
xor			

除上面的关键字外，**private**、**protected**、**public**、**published** 和 **automated** 在对象类型的声明中用作关键字，但其它情况下则作为指示字。关键字 **at** 和 **on** 也具有特殊的含义。

（以下内容摘自《Delphi 技术手册》
关键字是由 Delphi 编译器决定意义的保留标志符，不能把关键字用作变量、方法、或类型的名字等。）

Directives（指示字）

指示字只在代码中的特定位置才有特殊意义。在 Object Pascal 中，指示字具有特殊的意义，但和关键字不同，它只用于（用户）自定义标志符不能出现的上下文环境中。因此，你可以定义一个和指示字完全相同的标志符，虽然这是不明智的。

（以下内容摘自《Delphi 技术手册》
指示字是指在一个特定的上下文环境中，对编译器具有特殊意义的一个标志符。在上下文环境之外，你可以自由地把指示字的名字作为普通的标志符来使用。但是，编辑器并不总是对的，因为一些指示字的语法比较复杂，简单的编辑器不能正确处理。）

Numerals（数字）

整数和实数常量可以用十进制的阿拉伯数字序列来表示，数字之间不能有逗号或空格，但它可以有一个前缀“+”或“−”来表示正负。它的数值默认为正（所以，67258 和+67258 是相等的），并且必须位于预先定义的实数或整数类型的最大值之内。

带有小数点或指数符号的数字表示实数，而其它数字表示整数。当 E 或 e 出现在实数中时，它表示 10 的几次方。比如，7E−2 表示 7*10^−2，12.25e+6 和 12.25e6 都表示 12.25*10^6。

$前缀表示一个 16 进制数字，比如$8F。没有“−”号运算符的数字被认为是正数。在赋值时，若它的值在接收者的数据类型范围之外，将产生一个错误，除非它是整数类型（32 位整数），此时将引发一个警告，并且，values exceeding the positive range for Integer are taken to be negative numbers in a manner consistent with 2's complement integer representation.

关于实数和整数类型的更多信息，请参考数据类型、变量和常量。关于数字常量的数据类型，请参考真常量。

Labels（标签）

标签是一个不超过 4 位的阿拉伯数字序列，也就是从 0 到 9999，打头的 0 没有意义。标志符也可行使标签的功能。

标签用于 **goto** 语句中。关于 **goto** 语句的更多信息，请参考 Goto 语句。

Character strings（常量字符串）

常量字符串（character string）也称为文字串（string literal）或串常量（string const），它由引用串（由一对单引号括起来的文字串）、控制串（控制符构成的串）或这两种串的组合而构成。只有引用串可以包含分隔符。

引用串由扩展 ASCII 字符集的字符所组成，最多可达 255 个。它要书写在一行中，并且用一对单引号括起来。若单引号中没有内容（''），它被称为空串（null string）。在一个引用串中，两个连续的单引号（''）表示一个字符，也就是单引号本身（'）。看以下的例子：

```
'BORLAND'      { BORLAND }
'You''ll see'  { You'll see }
''''           { ' }
''             { 空串 }
' '            { 一个空格 }
```

控制串由一个或多个控制字符（控制符）所组成，每个控制符包含一个#，后跟一个无符号整数（10 进制或 16 进制），整数的范围从 0 到 255，表示相应的 ASCII 字符。下面的控制串

```
#89#111#117
```

就相当于引用串

```
'You'
```

你可以组合引用串和控制串来构成一个更大的串。例如，你能使用

```
'Line 1'#13#10'Line 2'
```

它在字符串 "Line 1" 和 "Line 2" 之间放一个回车（#13）换行（#10）符。但你不能使用这种方式组合两个引用串，因为两个连续的单引号被解释为一个单引号。要组合多个引用串，可以使用"+"运算符，或简单地把它们合并成一个引用串。

常量字符串的长度是它所包含的字符个数。一个任意长度的字符串，与任何字符串（string）类型以及 PChar 类型是兼容的；一个长度为 1 的字符串，与任何字符（character）类型兼容；并且，当启用扩展语法时（**{$X+}**），一个长度为 n 的非空字符串，和下标从 0 开始、包含 n 个字符的数组以及压缩（packed）数组也是兼容的。关于字符串类型的更多信息，请参考字符串类型。

Comments and compiler directives（注释和编译器指示字）

注释将被编译器忽略，除非它们用作分隔符（隔开相邻的 token）或编译器指示字。

有以下几种方式创建注释：

{ 由一对花括号所包含的文字构成注释 **}**
(* 由左圆括号加一个星号和
一个星号加右圆括号之间的文字也构成注释 ***)**
// 由两个斜杠开始直到这一行的结束，这里的文字是注释

若 **$** 符紧跟在 **{** 或 **(*** 之后，则这里的注释是编译器指示字。例如

```
{$WARNINGS OFF}
```

它告诉编译器不要产生警告信息。

Expressions（表达式）

About expressions（关于表达式）

表达式是一个有返回值的语句构造。比如，

```
X                     { 变量 }
@X                    { 变量地址 }
15                    { 整数常量 }
InterestRate          { 变量 }
Calc(X,Y)             { 函数调用 }
X * Y                 { X 和 Y 的乘积 }
Z / (1 - Z)           { Z 和(1 - Z)的商 }
X = 1.5               { 布尔 }
C in Range1           { 布尔 }
not Done              { 布尔的否 }
['a','b','c']         { 集合 }
Char(48)              { 类型转换 }
```

最简单的表达式是变量和常量（在数据类型、变量和常量中讲述）。更复杂的表达式由简单表达式使用运算符、函数调用、集合构造器、索引和类型转换构成。

Operators（运算符）

About operators（关于运算符）

运算符就像 Object Pascal 的内置函数，它是语言的一部分。例如，表达式（X+Y）由变量 X 和 Y（X、Y 称为运算数或操作数，operand，）通过“+”运算符计算而得。当 X 和 Y 表示整数或实数时，（X+Y）返回它们的和。运算符包括：@ **not** ^ * / **div** **mod** **and** **shl** **shr** **as** + – **or** **xor** = > < <> <= >= **in** 和 **is**。

@、**not** 和 ^ 是一元运算符（使用一个运算数），+ 和 – 或者是一元的，或者是二元的，除此之外，其它所有运算符为二元运算符（使用两个运算数）。除了 ^ 运算符，它在运算数的后面（比如，P^），其它一元运算符总是位于运算数的前面（比如，–B）。二元运算符位于运算数的中间（比如，A=7）。

一些运算符的行为因为传给它们的数据类型不同而不同。比如，**not** 运算符用于整数时，是对它的位进行求反运算，而对布尔类型进行逻辑非运算。这类运算符会在后面的多个分类中出现。

除了 ^ ，**is** 和 **in**，其它运算可应用在 Variant 类型上。

接下来的章节假设你对 Object Pascal 的数据类型有一定了解。

关于表达式中运算符的优先级，请参考运算符优先级一节。

Arithmatic operators（算术运算符）

算术运算符作用于实数或整数，包括 + 、− 、* 、 /、 **div** 和 **mod**。

运算符	运算	运算数类型	返回类型	例子
+	加	整数，实数	整数，实数	X + Y
−	减	整数，实数	整数，实数	Result - 1
*	乘	整数，实数	整数，实数	P * InterestRate
/	实数除	整数，实数	实数	X / 2
div	整数除	整数	整数	Total **div** UnitSize
mod	余数	整数	整数	Y **mod** 6

运算符	运算	运算数类型	返回类型	例子
+（一元）	正	整数，实数	整数，实数	+7
−（一元）	负	整数，实数	整数，实数	-X

以下规则适用于算术运算符：

- 不管 *x* 和 *y* 的类型是什么，*x*/*y* 的结果总是扩展类型（extended）；对其它运算符，只要有一个运算数是实数类型，它的结果就是扩展类型；另外，只要有一个运算数是 Int64 类型，它的结果就是 Int64 类型；否则，结果就是整数类型。如果一个运算数是整数的子界类型，它就像整数类型一样被对待。
- *x* **div** *y* 的值取整数，也就是取得 *x*/*y* 的值，然后以 0 的方向取得最近的整数。
- **mod** 运算返回对运算数进行整数除后得到的余数。换句话说，就是 *x* **mod** *y* = *x* − (*x* **div** *y*)**y*。
- 若 *y* 为 0 的话，表达式 *x*/*y*、*x* **div** *y* 和 *x* **mod** *y* 将发生运行时错误。

Boolean operators（布尔运算符）

布尔运算符 **not**、**and**、**or** 和 **xor** 作用于任何布尔类型的运算数，并返回一个布尔类型的值。

运算符	运算	运算数类型	结果类型	例子
not	否	布尔	布尔	**not** (C in MySet)
and	与	布尔	布尔	Done **and** (Total > 0)
or	或	布尔	布尔	A **or** B
xor	异或	布尔	布尔	A **xor** B

这些运算遵循标准的布尔逻辑规则。比如，像 *x* **and** *y* 形式的表达式，当且仅当 *x* 和 *y* 都为 True 时，它的结果才为 True。

Complete versus short-circuit Boolean evaluation（完全计算和部分计算）

编译器对 **and** 和 **or** 运算符支持两种计算方式：完全计算（complete evaluation）和部分计算（short-circuit evaluation 或 partial evaluation）。

完全计算会计算每个连接项（conjunct 和 disjunct）的值，即使整个表达式的结果已经确定了。部分计算从左到右计算每个连接项，一旦整个表达式的结果确定下来，计算就停止了。比如，对于表达式 *A*

and *B*，在进行部分计算时，若 *A* 是 False，则整个表达式的值也是 False，此时编译器不再计算 *B* 的值。

通常，部分计算更可取，因为它的执行时间最少，并且在大多数情况下使用最少的代码。当运算数是一个进行边界操作（side effects）的函数并影响程序的执行时，完全计算有时是比较方便的。

部分计算也能避免一些在其它情况下会产生的非法运行时错误。比如，下面的代码遍历字符串 S，直到发现第一个逗号

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  ...
  Inc(I);
end;
```

在 S 不包含逗号的情况下，最后一次循环使 I 的值大于 S 的长度，当测试下一次循环时，若进行完全计算，读取 S[I]将产生运行时错误；相反，若进行部分计算，**while** 条件的第二部分（S[I] <> ','）不会进行，因为前面的结果已经不成立了。

使用 **$B** 编译器指示字控制计算方式，默认状态是 **{$B-}**，它采用部分计算。要在局部进行完全计算，在代码中使用 **{$B+}** 指示字。你也可以在 Compiler Options 对话框中选择 Complete Boolean Evaluation 选项，此时在整个项目范围使用完全计算。

注意：若任何一个运算数是 **variant** 类型，编译器总是进行完全计算（即使在 **{$B-}** 状态下）。

Logical (bitwise) operators（位逻辑运算符）

下面的位运算符对整数（运算数）的位进行处理。比如，若 X 存储的是 001101（二进制形式），Y 是 100001，语句

```
Z := X or Y;
```

把值 101101 赋给 Z。

运算符	运算	运算数类型	返回类型	例子
not	位反	整数	整数	**not** X
and	位与	整数	整数	X **and** Y
or	位或	整数	整数	X **or** Y
xor	位异或	整数	整数	X **xor** Y
shl	位左移	整数	整数	X **shl** 2
shr	位右移	整数	整数	Y **shr** I

下面的规则适用于位运算符

- 位反（**not**）运算的返回类型和运算数相同；
- 若 **and**、**or** 或 **xor** 的运算数都是整数类型，则它的返回类型是包含运算数所有可能的值、且范围最小的预定义（内置）整数类型；
- 运算 *x* **shl** *y* 和 *x* **shr** *y* 把 *x* 的值向左或向右移 *y* 个位，也就等同于 *x* 乘或除以 2^*y*（2 的 *y* 次方），返回类型和 *x* 相同。比如，若 *N* 存储的是 01101（10 进制的 13），那么 *N* **shl** 1 返回 11010（10 进制 26）。注意，*y* 的值被解释为对 *x* 所属类型大小（位数）进行模运算，比如，若 *x* 是一个 *integer*，*x* **shl** 40 被解释为 *x* **shl** 8，因为 *integer* 的大小是 32 位（4 字节），40 **mod** 32 等于 8。

String operators（字符串运算符）

关系运算符 =、<>、<、>、<= 和 >= 都能对字符串进行操作（参考关系运算符）。+ 运算符连接两个字符串。

运算符	运算	运算数类型	返回类型	例子
+	连接	字符串、字符以及 packed string	字符串	S + '.'

下面的规则适用于字符串连接

- + 运算符的运算数可以是字符串、packed string（packed arrays of type *Char*）或字符。但是，若其中一个运算数是宽字符（WideChar）类型，其它运算数必须是长字符串。
- + 运算符的返回结果和任何字符串类型是兼容的。但是，若运算数都是短字符串或字符，并且它们的组合长度大于 255，则返回结果取前面的 255 个字符。

Pointer operators（指针运算符）

关系运算符 <、>、<= 和 >= 能对 *PChar* 类型进行操作（参考关系运算符）。下面的运算符也能使用指针类型作为运算数。关于指针的更多信息，请参考指针和指针类型。

运算符	运算	运算数类型	返回类型	例子
+	指针加	字符指针，整数	字符指针	P + I
–	指针减	字符指针，整数	字符指针，整数	P - Q
^	pointer dereference	指针	指针的基础类型	P^
=	等于	指针	布尔	P = Q
<>	不等于	指针	布尔	P <> Q

^ 运算符 dereference 一个指针（取得指针所指的内容，如何翻译呢？），除了通用指针 *Pointer* 以外，它的运算数可以是任何指针类型，对于 *Pointer* 类型，在 dereference 之前必须进行类型转换。

只有 P 和 Q 指向相同的地址，P = Q 才是真（*True*），否则 P <> Q 为真。

你能使用 + 和 – 运算符来增加和减少一个字符指针的偏移量，也能使用 – 运算符来比较两个字符指针偏移量的差。它遵循以下规则

- 若 I 是一个整数，P 是一个字符指针，那么 P + I 是把 P 的地址加上 I，也就是一个指向 P 后面第 I 个字符处的指针（表达式 I + P 等同于 P + I）；P – I 是把 P 的地址减去 I，也就是一个指向 P 前面第 I 个字符处的指针。
- 若 P 和 Q 都是字符指针，那么 P – Q 计算 P 的地址（高地址）和 Q 地址（低地址）之差，也就是返回一个表示 P 和 Q 之间字符数目的整数。P + Q 没有意义。

Set operators（集合运算符）

下面的运算符以集合作为运算数：

运算符	运算	运算数类型	返回类型	例子
+	并集	集合	集合	Set1 + Set2
–	差集	集合	集合	S - T

*	交集	集合	集合	S * T
<=	小于等于（子集）	集合	布尔	Q <= MySet
>=	大于等于（超集）	集合	布尔	S1 >= S2
=	等于	集合	布尔	S2 = MySet
<>	不等于	集合	布尔	MySet <> S1
in	成员关系	序数，集合	布尔	A **in** Set1

以下规则适用于 +、– 和 * 运算符：

- 当且仅当序数（集合基础类型中的一个值）O 属于集合 X 或集合 Y（或同时属于 X 和 Y）时，O 属于 X + Y；当且仅当 O 属于 X 但不属于 Y 时，O 属于 X – Y；当且仅当 O 同时属于 X 和 Y 时，O 属于 X * Y。
- +、– 和 * 的运算结果属于集合类型 **set of** A..B，这里 A 是结果中的最小序数，B 是结果中的最大序数。

以下规则适用于 <=、>=、=、<> 和 **in** 运算符：

- 只有当 X（集合）中的每个成员也是 Y（集合）中的成员时，X <= Y 才为真；Z >= W 等同于 W <= Z；只有当 U（集合）和 V（集合）正好拥有相同的成员时，U = V 才为真，否则 U <> V 为真；
- 对于序数 O 和集合 S，只有当 O 是 S 的一个成员时，O **in** S 才为真。

Relational operators（关系运算符）

关系运算符用来比较两个运算数。=、<>、<= 和 >= 也用作集合运算符（参考集合运算符），= 和 <> 也用作指针运算符（参考指针运算符）。

运算符	运算	运算数类型	返回类型	例子
=	等于	simple, class, class reference, interface, string, packed string	布尔	I = Max
<>	不等于	simple, class, class reference, interface, string, packed string	布尔	X <> Y
<	小于	simple, string, packed string, PChar	布尔	X < Y
>	大于	simple, string, packed string, PChar	布尔	Len > 0
<=	小于等于	simple, string, packed string, PChar	布尔	Cnt <= I
>=	大于等于	simple, string, packed string, PChar	布尔	I >= 1

对大多数简单类型，比较运算非常容易理解。比如，只有 I 和 J 有相同的值，I = J 才是真，否则 I <> J 为真。下面的规则适用于关系运算符：

- 除了实数和整数能一起比较外，两个运算数必须是兼容的类型；
- 对字符串进行比较，是依据它的每个字符在扩展 ASCII 字符集中的顺序，字符类型被当作长度为 1 的字符串；
- 两个 packed string 要进行比较，它们必须具有相同数目的元素；一个具有 n 个元素的 packed string 与一个字符串比较时，它被看作长度为 n 的字符串；
- 只有当两个 *PChar* 指针都指向同一个字符数组的范围内时，<、>、<= 和 >= 运算符才能作用于它们；
- 运算符 = 和 <> 能以类或类引用类型作为运算数。当用于类类型时，= 和 <> 的计算规则与

指针一样，只有当 C 和 D 指向同一个实例对象时，C = D 为真，否则 C <> D 为真；当用于类引用时，只有当 C 和 D 表示同一个类时，C = D 为真，否则 C <> D 为真。关于类的更多信息，请参考类和对象。

Class operators（类运算符）

as 和 **is** 运算符使用类和对象（实例）作为运算数，**as** 也用于接口类型。关于更多信息，请参考类和对象以及对象接口。

关系运算符 = 和 <> 也用于类类型，请参考关系运算符。

The @ operator（@运算符）

@ 运算符返回一个变量、函数、过程或方法的地址，也就是说，@ 运算符构建一个指向运算数的指针。关于指针的更多信息，请参考指针和指针类型。下面的规则适用于 @ 运算符：

- 若 X 是一个变量，@X 返回 X 的地址（当 X 是一个过程类型的变量时有特殊的规则，请参考语句和表达式中的过程类型）。若默认的编译器指示字 **{$T–}** 在起作用，则@X 的类型是 *Pointer*（通用指针）；在 **{$T+}** 状态下时，@X 的类型是 ^T，这里 T 是 X 的类型；
- 若 F 是一个例程（一个函数或过程），@F 返回 F 的入口点，@F 的类型总是 *Pointer*；
- 当 @ 作用于类的方法时，必须使用类名来限定方法名。比如
  @TMyClass.DoSomething
  它指向 *TMyClass* 的 *DoSomething* 方法。关于类和方法的更多信息，请参考类和对象。

Operator precedence rules（运算符优先级）

在复杂表达式中，运算符优先级规则决定了运算执行的顺序。

运算符	优先级
@，**not**	第一级（最高）
*，/，**div**，**mod**，**and**，**shl**，**shr**，**as**	第二级
+，–，**or**，**xor**	第三级
=，<>，<，>，<=，>=，**in**，**is**	第四级（最低）

具有较高优先级的运算符先进行运算，具有相同优先级的运算符从左边开始。因此表达式

X + Y * Z

先执行 Y*Z，然后加上 X 作为结果，* 先被执行，因为它比 + 有较高优先级，但

X - Y + Z

先从 X 减去 Y，然后加上 Z 作为结果，– 和 + 具有相同优先级，因此左边的运算先执行。

你能使用圆括号来覆盖优先级规则。有括号的表达式先被运算，然后把它作为单个运算数。比如

(X + Y) * Z

它把 X 和 Y 的和乘以 Z。

括号有时用于一些似是而非的场合。比如，考虑表达式

```
X = Y or X = Z
```

这里的明显用意是

```
(X = Y) or (X = Z)
```

但若没有括号，编译器遵循优先级规则把它读作

```
(X = (Y or X)) = Z
```

此时，若 Z 不是布尔类型，它将导致编译错误。

括号通常使代码更容易读写，即使有时候严格说来它们是多余的。这样，上面第一个例子可写作

```
X + (Y * Z)
```

这里，括号（对编译器）不是必须的，但对程序的编写者和阅读者来说，它节省了判断运算符优先级的时间。

Function calls（函数调用）

因为函数返回一个值，所以函数调用是表达式。比如，若你定义了一个叫做 Calc 的函数，它接收两个整数参数并返回一个整数，那么函数调用 Calc(24, 47)是一个整数表达式。若 I 和 J 是整数变量，那么 I + Calc(J, 8)也是整数变量。函数调用的例子包括

```
Sum(A, 63)
Maximum(147, J)
Sin(X + Y)
Eof(F)
Volume(Radius, Height)
GetValue
TSomeObject.SomeMethod(I,J);
```

关于函数的更多信息，请参考 Procedures and functions。

Set constructors（集合构造器）

集合构造器表示一个集合类型的值。比如，

```
[5, 6, 7, 8]
```

它表示一个成员是 5、6、7 和 8 的集合。集合构造器

```
[ 5..8 ]
```

它也能表示同一个集合。

集合构造器的语法是

[*item1*, ..., *itemn*]

这里，每个 *item* 或者是表示集合基础类型中的一个有序值（表达式），或者是由两个点（**..**）连接起来的一对这样的值（表达式）。若一个条目（item）是 x..y 的形式，它是从 x 到 y 范围间（包括 x 和 y）所有的有序值的简写。但若 x 比 y 大，则 x..y 不表示任何内容，**[x..y]**是空集。集合构造器**[ ]**表示空集，而**[x]**表示一个集合，它仅有的一个成员是 x 值。

集合构造器的例子：

```
[red, green, MyColor]
[1, 5, 10..K mod 12, 23]
['A'..'Z', 'a'..'z', Chr(Digit + 48)]
```

关于集合的更多信息，请参考 Sets。

Indexes（索引）

字符串、数组、数组属性以及指向字符串或数组的指针能被索引。比如，若 FileName 是一个字符串变量，表达式 FileName[3]返回 FileName 字符串中的第 3 个字符，而 FileName[I + 1]返回被 I 索引的后一个字符。关于字符串的信息，请参考 String types；关于数组和数组属性的信息，参考 Arrays 和 Array properties。

Typecasts（类型转换）

Typecasts: Overview（概述）

有时，把一个表达式当作一种不同的类型是很有用的。实际上，强制类型转换使你临时改变一个表达式的类型。比如，Integer('A')把一个字符 A 转换为一个整数。
强制类型转换的语法是

typeIdentifier(*expression*)

若表达式是一个变量，结果被称为 variable typecast（变量转换）；否则，结果是一个 value typecast（值转换）。虽然它们的语法相同，但它们有不同的转换规则。

Value typecasts（值转换）

在值转换中，类型标志符和转换的表达式必须都是有序类型或指针类型。值转换的例子包括

```
Integer('A')
Char(48)
Boolean(0)
Color(2)
Longint(@Buffer)
```

得到的结果是转换括号内的表达式。若指定的类型和表达式的的类型大小不同，结果会截断或扩展。表达式的符号总是被保留（这是什么意思？）。
语句

```
I := Integer('A');
```

把 Integer('A')的值（也就是 65）赋给变量 I。
一个值转换的后面不能有限定符（什么意思？），并且不能出现在赋值语句的左边。

Variable typecasts（变量转换）

倘若它们的大小（占用的内存）相同，并且你没有混用整数和实数，则可以把任何变量转换为任何类型（要转换数字类型，依靠标准函数，比如 Int 和 Trunc）。变量转换的例子包括

```
Char(I)
```

```
Boolean(Count)
TSomeDefinedType(MyVariable)
```

变量转换可出现在赋值语句的任何一边。这样

```
var MyChar: char;
...
Shortint(MyChar) := 122;
```

把字符 z（ASCII1 值是 122）赋给 MyChar。

你可以把变量转换为过程类型。比如，给出下面的声明

```
type Func = function(X: Integer): Integer;
var
  F: Func;
  P: Pointer;
  N: Integer;
```

你可以应用下面的赋值语句

```
F := Func(P);           { Assign procedural value in P to F }
Func(P) := F;           { Assign procedural value in F to P }
@F := P;                { Assign pointer value in P to F }
P := @F;                { Assign pointer value in F to P }
N := F(N);              { Call function via F }
N := Func(P)(N);        { Call function via P }
```

变量转换也可以跟限定符，像下面的例子所示

```
type
  TByteRec = record
    Lo, Hi: Byte;
  end;
  TWordRec = record
    Low, High: Word;
  end;
  PByte = ^Byte;
var
  B: Byte;
  W: Word;
  L: Longint;
  P: Pointer;
begin
  W := $1234;
  B := TByteRec(W).Lo;
  TByteRec(W).Hi := 0;
  L := $01234567;
  W := TWordRec(L).Low;
  B := TByteRec(TWordRec(L).Low).Hi;
  B := PByte(L)^;
end;
```

在这个例子中，TbyteRec 被用来访问一个字（word）的低字节和高字节，TwordRec 被用来访问一个长整数的低字和高字。你也可以使用内置函数 Lo 和 Hi 达到同样的目的，但变量转换的优点就是它可以用在赋值语句的左边。

要了解指针转换，请参考 Pointers and pointer types；要了解类和接口类型的转换，请参考 The as operator and Interface typecasts。

Declarations and Statements（声明和语句）

About Declarations and Statements（关于声明和语句）

除了 **uses** 子句（和划分单元不同部分的关键字，像 **implementation**），一个程序完全由声明和语句构成，声明和语句被组织成块（block）。

Declarations（声明）

变量、常量、类型、字段、属性、过程、函数、程序、单元、库和包的名称叫做标志符。（数字常量，像 26057，不是标志符）标志符在使用之前必须声明，唯一的例外就是一些内置的类型、例程和常量，还有函数块中的变量 Result，以及实现方法时的 Self 变量，编译器能自动识别它们。

一个声明定义一个标志符，并且在合适的地方为它分配内存。比如，

```
var Size: Extended;
```

声明一个叫做 Size 的变量，它存储一个 Extended（实数）值，而

```
function DoThis(X, Y: string): Integer;
```

声明一个叫做 DoThis 的函数，它接收两个字符串作为参数，并返回一个整数。每个声明以一个分号结束，当你同时声明多个变量、常量、类型或标签时，你只需书写合适的关键字一次就可以了。

```
var
  Size: Extended;
  Quantity: Integer;
  Description: string;
```

声明的语法以及声明的位置取决于要定义的标志符的种类。通常，声明只能出现在块（block）的开始处，以及单元的接口或实现部分的开始处（在 uses 子句之后）。声明变量、常量、类型、函数等的特殊约定在文档中的相关主题中解释。

Hint 指示字 **platform**、**deprecated** 和 **library** 能附加在任何声明之后。在声明过程或函数的情况下，应使用分号把 hint 指示字和声明的其它部分分开。比如：

```
procedure SomeOldRoutine; stdcall; deprecated;
var VersionNumber: Real library;
type AppError = class(Exception)
 ...
end platform;
```

当源代码在**{$HINTS ON} {$WARNINGS ON}**状态下编译时，对使用上述指示字声明的标志符的每个引用都将产生一个适当的提示或警告。使用 **platform** 标记一个条目和特定的操作系统（比如 Windows 和 Linux）相关；使用 **deprecated** 表示条目已经废弃或支持它仅为了向后兼容性；使用 **library** 表示依赖于特定的库或组件框架（比如 VCL 或 CLX）。

Statements（语句）

语句定义程序中的算法行为。简单语句，像赋值语句和过程调用，能组合成循环、条件语句以及其它结

构语句。

对于块中的多个语句、以及单元的初始化或结束化部分中的多个语句，使用分号把它们隔开。

Simple Statements（简单语句）

Simple Statements: Overview（概述）

一个简单语句不包含任何其它语句。简单语句包括赋值、过程和函数调用，以及 goto 跳转语句。

Assignment Statements（赋值语句）

赋值语句的格式如下

variable := *expression*

这里，*variable* 是任何变量引用，包括变量、变量类型转换、解除引用的指针，或者一个结构变量的组成部分；*expression* 是任何一个赋值兼容的表达式。（在函数块中，*variable* 能被函数名取代，参考 Procedures and functions。）符号 **:=** 有时叫做赋值运算符。

赋值语句使用 *expression* 的值取代 *variable* 的当前值。比如，

```
I := 3;
```

把 3 赋给变量 I。赋值语句左边的变量引用能出现在右边的表达式中。比如，

```
I := I + 1;
```

增加 I 的值。其它赋值语句的例子包括

```
X := Y + Z;
Done := (I >= 1) and (I < 100);
Hue1 := [Blue, Succ(C)];
I := Sqr(J) - I  * K;
Shortint(MyChar) := 122;
TByteRec(W).Hi := 0;
MyString[I] := 'A';
SomeArray[I + 1] := P^;
TMyObject.SomeProperty := True;
```

Procedure and Function Calls（过程和函数调用）

过程调用包含过程名（有或没有限定符），后面跟参数列表（若需要的话）。例子包括

```
PrintHeading;
Transpose(A, N, M);
Find(Smith, William);
Writeln('Hello world!');
DoSomething();
Unit1.SomeProcedure;
TMyObject.SomeMethod(X,Y);
```

当启用扩展语法时（**{$X+}**），调用函数也可以像调用过程那样，它被当作语句：

```
MyFunction(X);
```

当这样调用函数时，它的返回值被忽略。
关于过程和函数的更多信息，请参考 Procedures and functions。

Goto Statements（Goto 语句）

goto 语句格式如下

```
goto label
```

它把程序（执行）转移到指定标签所标记的语句。要标记一个语句，你必须首先定义这个标签，然后把这个标签和一个冒号放在语句的前面来标记它：

```
label: statement
```

像这样声明标签：

```
label label;
```

你能一次声明多个标签：

```
label label1, ..., labeln;
```

标签可以是任何有效标志符，也可以是 0 到 9999 之间的任何数值。
标签声明、标记的语句和 **goto** 语句必须属于同一个块（参考 Blocks and scope），因此，不能跳入一个过程或函数中，也不能从一个过程或函数中跳出。在一个块中，不能使用相同的标签标记多个语句。
比如，

```
label StartHere;
...
StartHere: Beep;
goto StartHere;
```

创建一个重复调用 Beep 过程的无限循环。
通常，在结构化编程中不鼓励使用 **goto** 语句，但有时使用它来退出嵌套循环，像下面的例子。

```
procedure FindFirstAnswer;
var X, Y, Z, Count: Integer;
label FoundAnAnswer;
begin
  Count := SomeConstant;
  for X := 1 to Count do
    for Y := 1 to Count do
      for Z := 1 to Count do
        if ... { some condition holds on X, Y, and Z } then
          goto FoundAnAnswer;
   ... {code to execute if no answer is found }
  Exit;
   FoundAnAnswer:
    ... { code to execute when an answer is found }
end;
```

注意：我们使用 **goto** 来跳出一个嵌套循环。永远不要跳入一个循环或其它结构语句，这会导致不可预知的结果。

Structured Statements（结构语句）

Structured Statements: Overview（概述）

结构语句由其它语句构成。当顺序执行其它语句，或有条件地、或重复执行其它语句时，使用结构语句。

- 复合语句或 **with** 语句只是简单地执行一系列语句；
- 条件语句，也就是 **if** 或 **case** 语句，根据指定的标准，最多执行一个分支；
- 循环语句，包括 **repeat**、**while** 和 **for** 循环，重复执行一系列语句；
- 一组特殊的语句，包括 **raise**、**try...except** 和 **try...finally** 结构，用来创建和处理异常。关于异常的产生和处理，请参考 Exceptions。

Compound Statements（复合语句）

一个复合语句由一系列其它语句（简单或结构语句）构成，它们的执行顺序和书写顺序一致。复合语句包含在关键字 begin 和 end 之间，构成它的语句由分号隔开。比如：

```
begin
  Z := X;
  X := Y;
  Y := Z;
end;
```

在 end 之前的最后一个分号是可选的，所以我们可以写作

```
begin
  Z := X;
  X := Y;
  Y := Z      // 这里能省略分号
end;
```

复合语句实际上用于 Object Pascal 语法要求有一个单一语句的地方。除了程序、函数和过程的块以外，它们还用于其它结构语句中，比如条件和循环语句。比如：

```
begin
  I := SomeConstant;
  while I > 0 do
  begin
    ...
    I := I - 1;
  end;
end;
```

在一个包含单一语句的地方，你可以使用复合语句，它就像复合条件中的括号。**begin** 和 **end** 有时用来增加可读性和消除歧义，你也能使用空复合语句创建一个块，它什么都不做。

```
begin
end;
```

With Statements（With 语句）

with 语句是一种简写方式，用来引用一个记录的字段，或一个对象的字段、属性和方法。with 语句的语法是

```
with obj do statement
```

或

```
with obj1, ..., objn do statement
```

这里，*obj* 是表示对象或记录的变量引用，*statement* 是任何简单或结构语句。在 *statement* 中，不用限定符、而仅使用 *obj* 的字段、属性和方法的名称就可以引用它们。

比如，给定声明

```
type TDate = record
  Day: Integer;
  Month: Integer;
  Year: Integer;
end;
var OrderDate: TDate;
```

你可以书写下面的 **with** 语句

```
with OrderDate do
  if Month = 12 then
  begin
    Month := 1;
    Year := Year + 1;
  end
  else
    Month := Month + 1;
```

这等同于

```
if OrderDate.Month = 12 then
begin
  OrderDate.Month := 1;
  OrderDate.Year := OrderDate.Year + 1;
end
else
  OrderDate.Month := OrderDate.Month + 1;
```

若 *obj* 涉及到索引数组或解除引用的指针，这个动作在 *statement* 之前执行一次。这使 **with** 语句既简洁又高效，也表明在 **with** 语句执行过程中，在 *statement* 中给一个变量赋值不会影响对 *obj* 的解释。

对 with 语句中的每个变量引用（字段或属性？）或方法名，尽可能把它解释为指定对象或记录的一个成员。若想在 **with** 语句中访问具有相同名称的其它变量或方法，你需要使用限定符，就像下面例子一样。

```
with OrderDate do
  begin
    Year := Unit1.Year
    ...
  end;
```

若在 **with** 后有多个对象或记录，则整个语句被看作是一系列嵌套的 **with** 语句，这样

```
with obj1, obj2, ..., objn do statement
```

等同于

```
with obj1 do
  with obj2 do
    ...
    with objn do
      statement
```

这种情况下，语句中的每个变量或方法名先尽可能被解释为 *obj*n 的成员，然后是 *obj*n-1 的成员，依此类推。对 *obj* 自身的解释也遵循同样的规则，所以举例来说，若 *obj*n 既是 *obj*1 的成员，又是 *obj*2 的成员，它被解释为 *obj*2.*obj*n。

If Statements（If 语句）

if 语句有两种形式：**if...then** 和 **if...then...else**。**if...then** 语句的语法是

```
if expression then statement
```

这里，*expression* 返回一个布尔值。若 *expression* 是 True，则 *statement* 被执行，否则它不执行。比如，

```
if J <> 0 then Result := I/J;
```

if...then...else 的语法是

```
if expression then statement1 else statement2
```

这里，*expression* 返回一个布尔值。若 *expression* 是 True，则 *statement*1 被执行，否则执行 *statement*2。比如，

```
if J = 0 then
  Exit
else
  Result := I/J;
```

then 和 **else** 子句每个（只）包含一个语句，但它可以是结构语句。比如，

```
if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end
else if Count = Last then
  Done := True
else
  Exit;
```

注意，在 **then** 子句和 **else** 之间不能有分号。你可以在整个 **if** 语句的后面放一个分号，把它和下一个语句隔开，但 **then** 和 **else** 子句除了一个空格或回车外，它不需要其它内容。在 **if** 语句中，**else** 后面的分号导致程序错误。

使用嵌套的 **if** 语句会产生解析困难，问题的出现是因为有些 **if** 语句有 **else** 子句，而另外一些则没有，但这两种语句的语法在其它方面是相同的。当嵌套语句中的 **else** 子句比 **if** 语句少时，就不容易判断哪个 **else** 子句和哪个 **if** 语句对应了。考虑下面形式的语句

```
if expression1 then if expression2 then statement1 else statement2;
```

这出现两种解析方式：

```
if expression1 then [ if expression2 then statement1 else statement2 ];
if expression1 then [ if expression2 then statement1 ] else statement2;
```

编译器总是按第一种方式解析它。用真的代码来表示的话，语句

```
if ... { expression1 } then
```

```
    if ... { expression2 } then
      ... { statement1 }
    else
      ... { statement2 } ;
```

等同于

```
  if ... { expression1 } then
  begin
    if ... { expression2 } then
      ... { statement1 }
    else
      ... { statement2 }
  end;
```

规则就是，对于嵌套的条件语句，解析从最内层开始，每个 else 对应于它左边最近的 if。对于我们的例子，要使编译器按第二种方式解析，你应该把代码明确写为

```
  if ... { expression1 } then
  begin
    if ... { expression2 } then
      ... { statement1 }
  end
  else
    ... { statement2 } ;
```

Case Statements（Case 语句）

相对于嵌套 **if** 语句的复杂性，**case** 语句具有更好的可读性。**case** 语句的形式是

```
  case selectorExpression of
    caseList1: statement1;
    ...
    caseListn: statementn;
  end
```

这里，*selectorExpression* 是任何一个有序类型的表达式（字符串无效），每个 *caseList* 是下列之一

- 数字、声明的常量或者编译器不需要执行程序就能计算的表达式。它必须是和 *selectorExpression* 兼容的有序类型。所以，7、True、4+5*3、'A'、以及 Integer('A')都能用于 *caseLists*，但变量和大多数函数不行（少数内置的函数，比如 Hi 和 Lo 能出现在 *caseList*，参考 Constant expressions）。
- 具有 *First*..*Last* 形式的子界类型，这里，*First* 和 *Last* 都满足上面的条件并且 *First* 小于或等于 *Last*。
- 具有 *item*1, ..., *item*n 形式的列表，这里，每个 *item* 满足上面的两个条件之一。

caseList 所表示的每个值必须是唯一的，子界类型和列表不能重叠。**case** 语句在最后能有一个 **else** 子句：

```
  case selectorExpression of
    caseList1: statement1;
    ...
    caseListn: statementn;
  else
    statements;
  end
```

这里，*statements* 是由分号隔开的语句序列。当 **case** 语句执行时，*statement*1 到 *statement*n 中最多有一个

执行，哪一个 *caseList* 和 *selectorExpression* 的值相等，哪个语句被执行。若 *caseList* 中没有一个和 *selectorExpression* 的值相等，那么 **else** 子句（若有的话）中的 *statements* 被执行。

case 语句

```
case I of
  1..5: Caption := 'Low';
  6..9: Caption := 'High';
  0, 10..99: Caption := 'Out of range';
else
  Caption := '';
end;
```

等同于下面的嵌套条件语句

```
if I in [1..5] then
  Caption := 'Low'
  else if I in [6..10] then
    Caption := 'High'
    else if (I = 0) or (I in [10..99]) then
      Caption := 'Out of range'
      else
        Caption := '';
```

其它 **case** 语句的例子有：

```
case MyColor of
  Red: X := 1;
  Green: X := 2;
  Blue: X := 3;
  Yellow, Orange, Black: X := 0;
end;
case Selection of
  Done: Form1.Close;
  Compute: CalculateTotal(UnitCost, Quantity);
else
  Beep;
end;
```

Control Loops（控制循环）

循环使你能重复执行一系列语句，它使用一个控制条件或变量来决定何时停止执行。Object Pascal 有三种循环：**repeat** 语句、**while** 语句和 **for** 语句。

你能使用 **Break** 和 **Continue** 过程来控制 **repeat**、**while** 或 **for** 语句的流程。**Break** 终止它所在的语句，而 **Continue** 开始执行下一次循环。

Repeat Statements（Repeat 语句）

repeat 语句的语法是

```
repeat statement1; ...; statementn; until expression
```

这里，*expression* 返回一个布尔值（**until** 之前的最后一个分号是可选的）。**repeat** 语句顺序执行它的命令

序列，在每次循环之后测试 *expression*，当 *expression* 返回 True 时，**repeat** 语句就停止。**repeat** 语句中的命令总是至少执行一次，因为直到完成第一次循环才测试 *expression*。

repeat 语句的例子包括

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;
repeat
  Write('Enter a value (0..9): ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

While Statements（While 语句）

while 语句和 **repeat** 语句类似，除了它的控制条件在第一次循环之前进行测试。因此，若条件为假，则命令永远不会执行。

while 语句的语法是

while *expression* **do** *statement*

这里，*expression* 返回一个布尔值，*statement* 可以是复合语句。**while** 语句重复执行构成它的命令，在每次循环前测试 *expression* 的值。只要 *expression* 返回 **True**，执行就继续下去。

while 语句的例子包括

```
while Data[I] <> X do I := I + 1;
while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;
while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

For Statements（For 语句）

for 语句不像 **repeat** 和 **while** 语句，它需要你明确指定循环进行的次数。**for** 语句的语法是

for *counter* := *initialValue* **to** *finalValue* **do** *statement*

或

for *counter* := *initialValue* **downto** *finalValue* **do** *statement*

这里

- *counter* 是一个有序类型的局部变量（在包含 **for** 语句的块中声明），没有任何限定符；
- *initialValue* 和 *finalValue* 是和 *counter* 赋值兼容的表达式；
- *statement* 是简单或结构语句，它不改变 *counter* 的值。

for 语句把 *initialValue* 的值赋给 *counter*，然后重复执行 *statement*，在每次循环后增加或减小 *counter* 的值（**for...to** 增加 *counter*，而 **for...downto** 减小 *counter*）。当 *counter* 的值和 *finalValue* 相同时，*statement* 再执行一次然后 **for** 语句终止。换句话说，对于 *initialValue* 到 *finalValue* 之间的每个值，*statement* 都执行一次。若 *initialValue* 等于 *finalValue*，*statement* 实际执行一次；若在 **for...to** 语句中 *initialValue* 比 *finalValue* 大，或在 **for...downto** 语句中 *initialValue* 比 *finalValue* 小，*statement* 永远不会执行。在 **for** 语句终止后，*counter* 值处于未知状态（未定义）。

为控制循环的执行，表达式 *initialValue* 和 *finalValue* 在循环开始前只执行一次。因此，**for...to** 近乎（但不）等于下面的 **while** 结构：

```
begin
  counter := initialValue;
  while counter <= finalValue do
  begin
    statement;
    counter := Succ(counter);
  end;
end
```

这个结构和 **for...to** 语句的不同之处在于，**while** 语句在每次循环之前要重新计算 *finalValue* 的值，若它是一个复杂表达式，这将明显降低执行速度，而且，在 *statement* 中改变 *finalValue* 的值会影响循环的执行。

for 语句的例子包括

```
for I := 2 to 63 do
  if Data[I] > Max then
    Max := Data[I];
for I := ListBox1.Items.Count - 1 downto 0 do
  ListBox1.Items[I] := UpperCase(ListBox1.Items[I]);
for I := 1 to 10 do
  for J := 1 to 10 do
  begin
    X := 0;
    for K := 1 to 10 do
      X := X + Mat1[I, K] * Mat2[K, J];
    Mat[I, J] := X;
  end;
for C := Red to Blue do Check(C);
```

Blocks and Scope（块和范围）

Blocks and Scope: Overview（概述）

声明和语句被组织成块，它为标签和标志符定义局部名字空间（或范围）。块能使一个单一的标志符（如变量名）在程序的不同部分有不同的意义。每个块是程序、函数或过程声明的一部分，每个程序、函数或过程声明有一个块。

Blocks（块）

一个块包含一系列的声明，后面跟一个符合语句。所有的声明必须一起出现在块的开始处，所以，块的形式是

```
declarations
begin
  statements
end
```

declarations 部分能声明变量、常量（包含资源字符串）、类型、过程、函数和标签，它们的顺序是任意的。在一个程序的块中，*declarations* 部分还能包含一个或多个 **exports** 子句（参考 Dynamic-link libraries and packages）。

比如，像下面的函数声明

```
function UpperCase(const S: string): string;
var
  Ch: Char;
  L: Integer;
  Source, Dest: PChar;
begin
  …
end;
```

声明的第一行是函数头，剩下的所有行构成了块。Ch、L、Source 和 Dest 是局部变量，它们的声明仅作用于 UpperCase 函数块中，并且覆盖（仅在这个块中）在程序块或单元的接口（或实现）部分对同一个标志符的声明。

Scope（范围）

一个标志符，比如变量或函数名，只能用于它声明的范围。声明的位置决定了它的范围，在声明程序、函数或过程时声明的标志符，它们的范围限是声明它的块；在单元的接口部分声明的标志符，它的范围包括使用它的任何其它单元或程序。具有较小范围的标志符，特别是函数和过程中声明的标志符，有时称为局部的（local）；具有较大范围的标志符称为全局的（global）。

决定标志符范围的规则如下：

若标志符声明出现在……	它的范围扩展到……
程序、函数或过程声明	从它声明的地方到当前块的末尾，包括当前块所包含的所有块。
单元的接口部分	从它声明的地方到单元的末尾，也包括使用这个单元的其它任何单元或程序。（参考 Programs and units）
单元的实现部分，但不是在任何函数或过程的块中。	从它声明的地方到单元的末尾，标志符对单元中的任何函数或过程是可用的，若有初始化和结束化部分的话，也包括它们。
记录类型的定义中（也就是说，标志符是记录的一个字段名）	从它声明的地方到记录类型定义的末尾（参考 Records）。
类的定义中（也就是说，标志符是类的字段、属性或方法名）	从它声明的地方到类定义的末尾，包括它的派生类；还包括类和派生类的所有方法块（参考 Classes and objects）。

命名冲突

当一个块包含另一个时，前者称为外部块，后者称为内部块。若外部块声明的标志符在内部块被重新声明，则内部声明覆盖外部声明，也决定了标志符在内部块中的意义。比如，若你在单元的接口部分声明

了变量 MaxValue，又在这个单元的一个函数中以同一个名字声明了另一个变量，在函数块中，没有限定的 MaxValue 指的是后者，是局部声明。类似的，当一个函数声明出现在另一个函数中时，它开辟了一个新的内部范围，在这里能重新声明被外部块使用的标志符。

使用多个单元令范围的定义复杂化。在 uses 子句中列出的每个单元，它引入（impose）一个新范围，这个范围包含 **uses** 子句中列在它之后的单元以及包含 **uses** 子句的程序或单元。在 **uses** 子句中最先列出的单元表示最外层（范围），每个接下来的单元表示前一个中的一个新范围。若两个或多个单元在接口部分声明了相同的标志符，没有限定的标志符引用使用最内层的声明，也就是说，是引用本身所在的单元，或者，若此单元没有声明它的话，则是 **uses** 子句中声明它的单元中的最后一个。

System 单元被每个程序或单元自动使用，它包含的声明，以及编译器自动解析的内置（预定义）类型、例程和常量，总是具有最外层范围。

通过使用限定符（参考 Qualified identifiers）或 **with** 语句（参考 With statements），你可以覆盖这些范围规则来绕过内层声明。

Data types, variables and constants（数据类型、变量和常量）

Data types and variables: Overview（概述）

类型在本质上是一种数据的名称。当声明一个变量时，必须指定它的类型；类型决定了它的取值范围和可以进行的操作（运算）。每个表达式返回一个特定类型的值，函数也是如此；大多数函数和过程要求指定类型的参数。

Object Pascal 是一种‘强类型’语言，也就是说，它对各种数据类型加以区分，并且不总是允许你用一种数据类型替代另一种。这通常是有好处的，因为它使编译器能聪明地处理数据并且更深入地验证你地代码，可以避免产生难于调试的运行时错误。但当你需要更多灵活性的时候，也有办法可以绕过强类型限制，它们包括强制类型转换（typecasting）、指针、变体类型（Variant）、记录中的变体部分（Variant parts in records）和绝对地址变量。

About types（关于类型）

有几种对 Object Pascal 数据类型进行分类的方法：

有些类型是内置的，编译器能自动识别，不必对它们进行声明。本语言参考中的几乎所有类型都是内置的；其它类型要通过声明来创建，它们包括用户自定义的类型以及在产品库（是 VCL 库吗？）中定义的类型。

类型可以分为基本（fundamental）和一般（generic）类型。在 Object Pascal 的实现上，基本类型的范围和形式是相同的，不管是基于何种 CPU 和操作系统；而一般类型的范围和形式是平台相关的，因实现的不同可能发生改变。大多数内置类型属于基本类型，但少数整数、字符、字符串和指针类型属于一般类型。在需要的时候使用一般数据类型是一个好注意，因为它们提供优化的性能和轻便性。但是，在不同的（语言）实现中，对于它们（一般类型数据）存储格式的改变会导致兼容性问题，比如，你向一个文件写入流数据。

类型也可以分为简单类型、字符串类型、结构类型、指针类型、过程类型和变体类型。另外，类型标志符本身也可以认为属于一种特殊‘类型’，因为它们可以作为参数传给一些函数（比如 High、Low 和 SizeOf）。

The outline below shows the taxonomy of Object Pascal data types.

下面的提纲显示了 Object Pascal 数据类型的分类：

- **simple**
  - ordinal
    - integer
    - character
    - Boolean
    - enumerated
    - subrange
  - real
- **string**
- **structured**
  - set

array
record
file
class
class reference
interface
pointer
procedural
Variant
（type identifier）
标准函数 **SizeOf** 作用于所有变量和类型标志符，它返回一个整数，表明存储指定类型的数据所需要的内存数（字节）。比如，SizeOf（Longint）返回 4，因为一个 Longint 变量使用 4 个字节的内存。
类型声明在以下章节说明。关于类型声明的一般信息，请参考 Declaring types。

（以下内容摘自《**Delphi** 技术手册》
有些类型是在编译器中内置的，但更多的是在 System 单元中明确定义的）

Simple types（简单类型）

Simple types: Overview（概述）

简单类型包括有序类型和实数类型，它们定义有次序的数值集合。

Ordinal types（有序类型）

Ordinal types: Overview（概述）

有序类型包括整数、字符、布尔、枚举和子界类型。有序类型定义一个有次序的数值集合，除了它的第一个值以外，其它每个值都有一个唯一的前驱值（predecessor）；除了最后一个外，其它每个值都有一个唯一的后继值（successor）。并且，每个值都有一个序数决定它在这个类型中的位置。在大多数情况下，如果一个值的序数为 n，它的前驱值序数为 n−1，它的后继值序数为 n+1。

- 对整数类型，一个值的序数为它本身的值
- 子界类型保留它们的基础类型的序数值
- 对其它有序类型，默认情况下，第一个值的序数为 0，下一个为 1，依此类推。声明一个枚举类型时，可明确地覆盖默认值。

有几个内置的函数作用于有序类型的数据和类型标志符，下面是最重要的几个：

函数	参数	返回值	说明
Ord	有序类型表达式	序数值	不能用于 Int64 类型
Pred	有序类型表达式	表达式的前驱值	
Succ	有序类型表达式	表达式的后继值	
High	有序类型的变量或标志符	类型的最大值	可用于短字符串或数组
Low	有序类型的变量或标志符	类型的最小值	可用于短字符串或数组

比如，High（Byte）返回 255，因为 Byte 类型的最大值是 255，Succ（2）返回 3，因为 3 是 2 的后继值。标准函数 Inc 和 Dec 分别增加和减少一个有序数据的值。例如，Inc（I）就相当于 Succ（I），而且，如果 I 是一个整数类型的话，也相当于 I := I + 1。

Integer types（整数类型）

整数类型表示所有数字的一个子集。一般（generic）整数类型是 Integer 和 Cardinal，只要可能就尽量使用它们，因为它们对依赖的 CPU 和操作系统作了优化。下表给出了 32 位编译器下它们的取值范围和存储格式：

类型	取值范围	格式
Integer	−2147483648..2147483647	32 位有符号
Cardinal	0..4294967295	32 位无符号

基本整数类型包括 Shortint、Smallint、Longint、Int64、Byte、Word 和 Longword。

类型	取值范围	格式
Shortint	−128..127	8 位有符号
Smallint	−32768..32767	16 位有符号
Longint	−2147483648..2147483647	32 位有符号
Int64	−2^63..2^63−1	64 位有符号
Byte	0..255	8 位无符号
Word	0..65535	16 位无符号
Longword	0..4294967295	32 位无符号

通常，对整数类型进行（算术）运算返回一个 Integer 类型（当前相当于 32 位的 Longint）。只有当对一个 Int64 类型的整数运算时才返回 Int64 类型。所以，下面的代码将产生错误：

```
var
  I: Integer;
  J: Int64;
  …
I := High(Integer);
J := I + 1;             （I+1 为 Integer 类型）
```

要在上面返回一个 Int64 类型的值，把 I 强制转换为 Int64 类型：

```
…
J := Int64(I) + 1;     （现在就是 Int64 类型了）
```

要得到更多信息，请参考 Arithmetic operators。

注意：大多数使用整型参数的标准例程，会把 Int64 数值截取为 32 位。但是，例程 **High**、**Low**、**Succ**、**Pred**、**Inc**、**Dec**、**IntToStr** 和 **IntToHex** 完全支持 Int64 参数。而且，**Round**、**Trunc**、**StrToInt64** 和 **StrToInt64Def** 返回 Int64 类型的值；少数例程（包括 Ord）根本不能对 Int64 数值进行操作。

当把整数类型的最后一个值增大或把第一个值减少时，结果将回到整数类型的开头或尾部。比如，Shortint 类型的取值范围是−128..127，所以，当下面的代码执行后

```
var I: Shortint;
 …
```

```
I := High(Shortint);
I := I + 1;
```

I 的值将是−128。但如果打开了编译器边界检查（range-checking），上面的代码将产生运行时错误。

Character types（字符类型）

基本字符类型是 AnsiChar 和 WideChar。**AnsiChar 是 8 位字符集，它们将依照本地字符集的顺序排列，这可能是多字节的**。AnsiChar 原先是根据 ANSI 字符集建立的，但现在扩展为可以指定本地字符集。WideChar 使用多于一个字节来表示每个字符。在当前实现中，WideChar 依据 Unicode 字符集（要意识到它的实现将来可能会改变），使用一个字的宽度（16-bit）来存储字符。开头的 256 个 Unicode 字符对应于 ANSI 字符。
一般字符类型是 Char，它相当于 AnsiChar。因为 Char 的实现可能被迫改变，所以，在需要处理不同大小的字符时，应该使用 SizeOf 而不要使用硬编码常数，这通常是个好主意。
一个长度为 1 的字符串常量，比如'A'，可以表示一个字符。内置函数 Chr，返回一个在 AnsiChar 或 WideChar 取值范围内任意一个整数的字符值。比如，Chr（65）返回字母 A。
字符和整数一样，当因为增加或减少而超过它的取值范围的开头或尾部时，它的值将回转（除非开启了边界检查）。比如，下面的代码执行后

```
var
  Letter: Char;
  I: Integer;
begin
  Letter := High(Letter);
  for I := 1 to 66 do
    Inc(Letter);
end;
```

Letter 的值将是 A（ASCII 值为 65）

Boolean types（布尔类型）

4 种内置的布尔类型为 Boolean、ByteBool、WordBool 和 LongBool，Boolean 是首选的，另外三种是为了和其它语言以及操作系统库相兼容。
一个 Boolean 类型的变量占据一个字节，ByteBool 也是，WordBool 类型占据两个字节（1 word），LongBool 占据 4 个字节（2 words）。
Boolean 值由内置的常数 True 和 False 来表示。

Boolean	**ByteBool, WordBool, LongBool**
False < True	False <> True
Ord(False) = 0	Ord(False) = 0
Ord(True) = 1	Ord(True) <> 0
Succ(False) = True	Succ(False) = True
Pred(True) = False	Pred(False) = True

对 ByteBool、WordBool 和 LongBool 来说，若它的序数不为 0，它就被认为是 True。在一个需要 Boolean 类型的环境种，编译器自动把非 0 值转换为 True。

前面说明指的是 Boolean 的序数值，而不是它们自身的值。在 Object Pascal 中，布尔表达式不能和整数或实数进行换算。所以，若 X 是一个整数变量，语句

```
if X then …;
```

会产生编译错误。把这个变量强制转换为布尔类型也是不可取的，但下面的用法可以正常工作：

```
if X <> 0 then …;        { use longer expression that returns Boolean value }
var OK: Boolean           { use Boolean variable }
 …
if X <> 0 then OK := True;
if OK then …;
```

Enumerated types（枚举类型）

枚举类型定义一个有次序的值的集合：这些值用标志符表示，并被列举出来，但它们并没有内在的含义。定义一个枚举类型，使用下面的语法：

```
type typeName = (val1, …, valn)
```

这里，*typeName* 和每个值是有效的标志符。例如，下面的声明

```
type Suit = (Club, Diamond, Heart, Spade);
```

定义了一个叫做 Suit 的枚举类型，它的可能值包括 Club、Diamond、Heart 和 Spade。这里，Ord（Club）返回 0，Ord（Diamond）返回 1，依此类推。

当定义一个枚举类型时，也就同时定义了它的每一个值：它是类型 *typeName* 的一个常量。如果在同一个范围内你使用 *val* 标志符用作其它目的，将产生命名冲突。例如，假设你声明类型：

```
type TSound = (Click, Clack, Clock);
```

不幸的是，Click 也是 TControl（以及它的子类）的一个方法名，所以，如果在程序中创建了如下的事件句柄：

```
procedure TForm1.DBGrid1Enter(Sender: TObject);
var Thing: TSound;
begin
  …
  Thing := Click;
  …
end;
```

将导致编译错误：编译器会认为这个过程中的 *click* 是 TForm 的 Click 方法。此时，你可以使用限定符来完成工作，这样，如果 TSound 是在 MyUnit 单元声明的，你可以使用

```
Thing := MyUnit.Click;
```

但一种更好的方式是使用不和其它标志符产生冲突的名字。例如：

```
type
  TSound = (tsClick, tsClack, tsClock);
  TMyColor = (mcRed, mcBlue, mcGreen, mcYellow, mcOrange);
  Answer = (ansYes, ansNo, ansMaybe);
```

在声明变量时，你可以直接使用（*val1*，…，*valn*）结构，它就像一个类型名称：

```
var MyCard: (Club, Diamond, Heart, Spade);
```

但以这种方式声明 MyCard，你就不能在同一范围，用这些常量标志符来声明其它变量了。这样

```
var Card1: (Club, Diamond, Heart, Spade);
var Card2: (Club, Diamond, Heart, Spade);
```

会产生编译错误，但是

```
var Card1, Card2: (Club, Diamond, Heart, Spade);
```

会很好地编译，下面的也是

```
type Suit = (Club, Diamond, Heart, Spade);
var
  Card1: Suit;
  Card2: Suit;
```

Enumerated types with explicitly assigned ordinality（指定序数值的枚举类型）

默认情况下，枚举类型的序数从 0 开始，并按它们的标志符在声明时列出的顺序排列。在声明时，通过给它的某些或所有值明确地指定一个序数，可以覆盖默认地序数。要给一个值赋序数，使用= *constantExpression* 紧跟在它的标志符之后，这里，*constantExpression* 为一个整数类型的常量表达式。比如

```
type Size = (Small = 5, Medium = 10, Large = Small + Medium);
```

定义了一个叫做 Size 的类型，它的可能值包括 Small、Medium 和 Large，这里 Ord（Small）返回 5，Ord（Medium）返回 10，Ord（Large）返回 15。

实际上，枚举类型是这样一个子界类型：它的最小值和最大值，分别对应于声明时的最小序数和最大序数。在上面的例子中，Size 类型的最小序数为 5，最大为 15，所以它有 11 个可能的值(因此，类型 **array**[Size] **of** Char 表示一个有 11 个元素的字符数组)。虽然只有其中的 3 个值有名称，但其它值通过类型转换、或通过一些例程（比如 Pred、Succ、Inc 和 Dec）是可以访问的。在下面的例子中，Size 取值范围内的‘无名’值被赋给 X：

```
var X: Size;
X := Small;       // Ord(X) = 5
X := Size(6);     // Ord(X) = 6
Inc(X);           // Ord(X) = 7
```

若一个值没有明确指定序数，它的取值将是前一个值的序数值加上 1；如果第 1 个值没有指定序数，它的序数为 0。所以，在下面的声明中：

```
type SomeEnum = (e1, e2, e3 = 1);
```

SomeEnum 只有两个可能的值：Ord（e1）返回 0，Ord（e2）返回 1，Ord（e3）也是 1。因为 e2 和 e3 有相同的序数，它们表示相同的值。

Subrange types（子界类型）

子界类型表示其它有序类型（称为基础类型）的一个子集：它的形式为 *Low*..*High*。这里，*Low* 和 *High* 是同一种有序类型的常量表达式，并且，*Low* 比 *High* 要小。以上面的形式，就定义了一个子界类型，它包括 *Low* 和 *High* 之间的所有值。比如，若你声明枚举类型：

```
type TColors = (Red, Blue, Green, Yellow, Orange, Purple, White, Black);
```

你就可以这样定义一个子界类型：

```
type TMyColors = Green..White;
```

这里，TMyColors 就包含值 Green、Yellow、Orange、Purple 和 White。

你能使用数字常量和字符（长度为 1 的字符串）定义子界类型：

```
type
  SomeNumbers = -128..127;
  Caps = 'A'..'Z';
```

当使用数字或字符常量定义一个子界类型时，基础类型是最小的整数类型或包含指定区间的最小字符类型（由编译器决定）。

Low..High 构造本身就相当于一个类型名称，所以，你可以直接使用它来声明变量。例如：

```
var SomeNum: 1..500;
```

定义了一个整型变量，它可以是 1 到 500 中的任何值。

对子界类型，每个值的序数取自基础类型（在上面的第一个例子中，若 Color 变量的值为 Green，Ord（Color）将返回 2，而不管它是 TColors 类型或 TMyColors 类型）；而且，它的值不会回转到开头或结尾，即使它的基础类型是整型或字符类型；当因增大或减小而产生越界时，它的值只是简单地转换成基础类型的值。因此，虽然

```
type Percentile = 0..99;
var I: Percentile;
…
I := 100;
```

产生一个错误，但

```
…
I := 99;
Inc(I);
```

把值 100 赋给 I（除非打开编译器的边界检查功能）。

在子界类型的定义中，常量表达式的使用会使语法分析出现困难。在任何类型的声明中，当'＝'后面的第一个（有意义的）字符是左圆括号时，编译器就假定是在定义一个枚举类型。因此，下面的代码

```
const
  X = 50;
  Y = 10;
type
  Scale = (X - Y) * 2..(X + Y) * 2;
```

会产生错误。你可以去掉开头的圆括号解决这个问题

```
type
  Scale = 2 * (X - Y)..(X + Y) * 2;
```

Real types（实数类型）

实数类型定义了一类可以用浮点表示的数字。下表给出了基本实数类型的范围和存储格式：

类型	范围	有效位数	字节大小
Real48	2.9 x 10^-39 .. 1.7 x 10^38	11-12	6
Single	1.5 x 10^-45 .. 3.4 x 10^38	7-8	4
Double	5.0 x 10^-324 .. 1.7 x 10^308	15-16	8
Extended	3.6 x 10^-4951 .. 1.1 x 10^4932	19-20	10
Comp	-2^63+1 .. 2^63-1	19-20	8
Currency	-922337203685477.5808.. 922337203685477.5807	19-20	8

一般实数类型为 Real，在当前实现中，它相当于 Double。

注意：在 Object Pascal 的早期版本中，Real 相当于 6 字节的 Real48，若要重新编译以前的代码，你可能要把它改为 Real48；你也可以使用{$REALCOMPATIBILITY ON}编译器指示字把 Real 类型转回到原先类型。

以下备注适用于基本实数类型：

- 保留 Real48 是为了向后兼容性。因为它的存储格式在 Intel CPU 下不能优化，这将导致比其它浮点

类型运行稍慢。

- Extended 类型比其它实数类型有更高的精度，但不够轻巧。当使用 Extended 类型创建的文件要跨平台共享时，务必小心。
- Comp（computational）类型对 Intel CPU 是优化的，表示为 64 位整数，但它被分类位实数类型，因为它的行为不像有序类型（比如，你不能递增和递减）。保留此类型只是为了向后兼容性，使用 Int64 可获得更好的性能。
- Currency 类型有固定的小数点，可以把在货币运算中出现的错误减到最少。It is stored as a scaled 64-bit integer with the four least significant digits implicitly representing decimal places. 当在赋值语句和表达式中与其它实数类型混合使用时，它的值会自动除以或乘上 10000。

String types（字符串类型）

About string types（关于字符串类型）

字符串表示一个字符序列。Object Pascal 支持以下种类的内置字符串：

类型	最大长度	所需内存（字节）	用于
ShortString	255 个字符	2 － 256	向后兼容
AnsiString	~2^31 个字符	4 － 2GB	8 位（ANSI）字符
WideString	~2^30 个字符	4 － 2GB	Unicode 字符 multi-user servers and multi-language applications

AnsiString，有时称为长字符串，在大多数情况下是首选类型。

在赋值语句和表达式中，字符串类型可以混合使用，编译器自动进行所需的转换。但对于过程或函数，当使用引用方式传递字符串时，类型必须正确。字符串可明确地被转换为其它类型的字符串。

关键字 **string** 用起来就像一个一般类型名，例如

```
var S: string;
```

声明一个变量 S，它保存字符串类型。在默认的**{$H+}**状态下，编译器把 **string**（当它的后面没有包含数字的中括号时）解释为 AnsiString；使用**{$H−}**指示字把它解释为 ShortString。

标准函数 Length 返回一个字符串的字符个数；SetLength 设置一个字符串的长度。

对字符串的比较，是比较它们对应位置上的字符大小（顺序）。对长度不等的字符串，较长字符串上多余的字符被认为较大（若前面没有比较出大小）。例如，'AB'比'A'大，也就是说，'AB' > 'A'返回 True。零长度的字符串有最小值。

你可以像数组一样对字符串使用索引。若 S 是一个字符串变量，i 是一个整数表达式，则 S[i]表示 S 中第 i 个字符（或者，严格说来，是第 i 个字节）。对于 ShortString 或 AnsiString，S[i]是 AnsiChar 类型；对于 WideString，S[i]是 WideChar 类型。语句 `MyString[2] := 'A';` 把值 A 赋给 MyString 的第 2 个字符。

下面的代码使用 UpCase 函数把 MyString 转换为大写：

```
var I: Integer;
begin
  I := Length(MyString);
  while I > 0 do
  begin
    MyString[I] := UpCase(MyString[I]);
    I := I - 1;
  end;
```

```
end;
```

像这样对字符串使用索引要非常小心，因为重写一个字符串的结尾时会导致访问违规。而且，要避免传递长串的索引作为 var 参数，因为这样会降低效率。

你可以把一个字符串常量（或返回一个字符串的表达式）赋给一个变量，在赋值发生时，字符串的长度能动态改变。比如：

```
MyString := 'Hello world!';
MyString := 'Hello ' + 'world';
MyString := MyString + '!';
MyString := ' ';                    { space }
MyString := '';                     { empty string }
```

Short strings（短字符串）

一个 ShortString 可包含 0 到 255 个字符。它的长度能动态改变，它被**静态分配 256 字节的内存**：第 1 个字节存储串的长度，剩下的 255 个字节存储字符。若 S 是一个 ShortString 变量，Ord（S[0]），和 Length（S）一样，将返回 S 的长度；给 S[0]赋值，就像调用 SetLength，将改变 S 的长度。ShortString 使用 8 位 ANSI 字符，保留它只是为了向后兼容性。

Object Pascal 支持 short-string 类型（实际上，它是 ShortString 的子类型），它的最大长度可以是从 0 到 255 之间的任何值。它通过在保留字 **string** 的后面添加一对包含数字的中括号来声明。比如

```
var MyString: string[100];
```

声明一个叫做 MyString 的变量，它的最大长度是 100 字节，这和以下的声明效果相同

```
type CString = string[100];
var MyString: CString;
```

像这样声明的变量，它们只分配所需的内存，也就是指定的最大长度加上一个字节。在我们的例子中，MyString 使用 101 个字节，相比之下，使用内置的 ShortString 类型将分配 256 个字节。

当给一个 short-string 变量赋值时，多于它最大长度的部分将被截取掉。

标准函数 High 和 Low 能作用于 short-string 类型名和变量，High 返回它的最大长度，Low 返回 0。

Long strings（长字符串）

AnsiString 类型又称为长字符串，它可以动态分配，并且长度只受内存限制。它使用 8 位 ANSI 字符。

长串变量是一个指针，占据 4 个字节的内存。当变量为空时（也就是长度为 0 的字符串），指针为 **nil**，此时，它不需要额外的内存；当变量为非空时，它指向一个动态分配的内存块，内存块存储字符串的值：一个 32 位的长度指示器，一个 32 位的引用计数器。**它的内存在堆中分配**，但它的管理是完全自动的，不需要自己编写代码。

因为长串变量是指针，所以，两个或更多的变量可以引用同一个值，而不必使用额外的内存。编译器利用这一点节省资源和进行快速赋值。只要一个长串变量被销毁或赋给一个新值，原来的串（变量的前一个值）引用计数减 1，而新的值（如果有的话）引用计数加 1。若引用计数为 0，它的内存被释放。这个过程被称为 reference-counting。当使用字符串索引改变其中的一个字符时，若字符串的引用计数大于 1，将生成串的一个拷贝，这被称为 copy-on-write 机制。

Wide strings（宽字符串）

WideString（宽字符串）

WideString 类型是动态分配的、由 16 位 **Unicode** 字符所构成的字符串。在大多数方面，它和 AnsiString 相似。（注：**宽字符串没有引用计数，不支持 copy-on-write 机制，但支持内存动态分配**。）
在 Win32 下，WideString 和 COM BSTR 类型兼容。Borland 开发工具支持把 AnsiString 类型转换为 WideString，但你可能需要明确地使用类型转换。

About extended character sets（关于扩展字符集）

Windows 和 Linux 都支持单字节和多字节字符集，同样也支持 Unicode。使用单字节字符集（SBCS），字符串的每个字节表示一个字符。ANSI 字符集被许多西方（国家的）操作系统使用，它是单字节字符集。
在多字节字符集（MBCS）中，一些字符由一个字节表示，而另一些则使用多个字节。多字节字符的第一个字节称为**头字节**（lead byte）。通常，在多字节字符集中，前 128 个字符对应于 7 位 ASCII 字符（0−127），**任何序数值大于 127 的字节为多字节字符的头字节**。只有单字节字符能包含空值（#0）。多字节字符集（特别是双字节字符集，DBCS），被亚洲语言广泛使用，但 Linux 使用的 UTF-8 字符集是 Unicode 的一种多字节编码。
在 Unicode 字符集中，每个字符用两个字节表示，所以，一个 Unicode 字符串是由双字节组成的序列。Unicode 字符和字符串也被称为宽字符和宽字符串。前 256 个 Unicode 字符被映射到 ANSI 字符集。Windows 操作系统支持 Unicode（UCS-2）。Linux 操作系统支持 UCS-4，是 UCS-2 的超集。Delphi/Kylix 在两个平台上都支持 UCS-2。
Object Pascal 支持单字节和多字节字符以及字符串，适用的类型有：Char、PChar、AnsiChar、PAnsiChar 和 AnsiString。对多字节字符串使用索引是不可取的，因为 S[i]表示 S 中第 i 个字节（不一定是第 i 个字符），但是，标准字符串处理函数有多字节版本，它们还实现了 locale-specific ordering for characters（多字节函数的名称通常以 Ansi 开头，比如 StrPos 的多字节版本是 AnsiStrPos）。多字节支持依赖于操作系统和本地设置（current locale）。
Object Pascal 使用 WideChar、PWideChar 和 WideString 类型支持 Unicode 字符和字符串。

Null-terminated strings（零结尾字符串）

Working with null-terminated strings（使用零结尾字符串）

许多编程语言，包括 C 和 C++，它们没有专门的字符串类型。这些语言以及它们创建的系统依赖于零结尾字符串，**它们是 0 下标开始的字符数组，并且最后一个是 NULL（#0）**。因为它们没有长度指示，第一个 NULL 字符就是字符串的结尾。当需要与其它语言创建的系统共享数据时，你可以使用 Object Pascal 句法以及 SysUtils 单元的特殊例程来处理零结尾字符串。
例如，下面的类型声明可以用于存储零结尾字符串：

```
type
  TIdentifier = array[0..15] of Char;
  TFileName = array[0..259] of Char;
```

```
TMemoText = array[0..1023] of WideChar;
```

当启用扩展语法时（{$X+}，这是默认的），**你可以把一个字符串常量，赋给一个 0 下标开始的静态字符数组**（动态数组此时不能工作）。当用字符串常量初始化一个字符数组时，若字符串的长度比数组声明的长度要短，其余的字符被设置为#0。（注：当用字符串常量给一个字符数组赋值时，结果也总是如此。）

Using pointers, arrays, and string constants（使用指针、数组和字符串常量）

要操作零结尾字符串，要经常需要指针。**字符串常量和类型 PChar、PWideChar 是赋值兼容的**，后两者表示指针，它们指向一个以 0 结尾的 Char 或 WideChar 字符数组。比如：

```
var P: PChar;
 …
P := 'Hello world!';     （注：编译器在字符串的末尾添加一个 NULL）
```

使 P 指向一个内存区域，在这里存储着'Hello world!' ，它的结尾被添加一个 NULL 字符，这和下面的效果相同：

```
const TempString: array[0..12] of Char = 'Hello world!'#0;
var P: PChar;
 …
P := @TempString;    （注：和 P := TempString 相同）
```

另外，对于使用 PChar 或 PWideChar 类型、并采用传值或常量参数的函数，你也可以给它传递字符串常量，比如 StrUpper（'Hello world!'）。当给 PChar 赋值时，编译器创建一个以 0 结尾的字符串的拷贝（generates a null-terminated copy of the string。注：其实只有一个字符串，PChar 变量是一个指向它的指针），并给函数传递一个指向这个拷贝的指针。当然，你也可以初始化一个 PChar 或 PWideChar 类型的常量，比如：

```
const
Message: PChar = 'Program terminated';
  Prompt: PChar = 'Enter values: ';
  Digits: array[0..9] of PChar = (
    'Zero', 'One', 'Two', 'Three', 'Four',
    'Five', 'Six', 'Seven', 'Eight', 'Nine');
```

0 下标开始的字符数组和 PChar 以及 PWideChar 兼容。当使用字符数组代替一个指针（值）时，编译器把数组转换为一个指针常量，它的值对应于数组的第一个元素的地址（也就是数组的地址）。比如

```
var
  MyArray: array[0..32] of Char;
  MyPointer: PChar;
begin
  MyArray := 'Hello';
  MyPointer := MyArray; （注：和@MyArray 相同）
  SomeProcedure(MyArray);
  SomeProcedure(MyPointer);
end;
```

上面的代码使用同一个值调用 SomeProcedure 两次。

一个字符指针（PChar 或 PWideChar）可以像数组一样使用索引。在上面的例子中，MyPointer[0]返回字符 H。索引实际上指定了一个偏移量（对 PWideChar 变量，索引自动乘以 2）。这样，如果 P 是一个字符指针，P[0]和 P^是相同的，它们指的都是数组中的第 1 个字符，P[1]是数组中的第 2 个字符，依此类推。P[-1]指的是紧靠 P[0]左边的‘字符’（注：这里可以是任何值），编译器对这些索引并不进行边界检查。

下面的 StrUpper 函数，演示了使用指针索引对一个零结尾字符串进行遍历操作：

```
function StrUpper(Dest, Source: PChar; MaxLen: Integer): PChar;
var
  I: Integer;
begin
  I := 0;
  while (I < MaxLen) and (Source[I] <> #0) do
  begin
    Dest[I] := UpCase(Source[I]);
    Inc(I);
  end;
  Dest[I] := #0;
  Result := Dest;
end;
```

Mixing Pascal strings and null-terminated strings（混合使用 Pascal 字符串和零结尾字符串）

在表达式和赋值语句中，你可以混合使用长字符串（AnsiString）和零结尾字符串（PChar），并且，对使用长字符串的函数或过程，你也可以通过 PChar 进行传值。赋值语句：S := P（这里，S 是一个字符串，P 是一个 PChar 表达式），把一个零结尾字符串拷贝到长字符串中。（注：长字符串是动态分配内存的，并在堆中分配，所以，它是把 P 指向的字符串复制到自己的内存区。）

在二元操作中，如果一个运算数是长字符串，而另一个是 PChar，PChar 将被转换为长字符串类型。（注：怎么转换呢？应该是复制一个吧。）

你可以把一个 PChar 值强制转换为长字符串，当要对两个 PChar 进行（长）字符串操作时，这就非常有用。比如：

```
S := string(P1) + string(P2);
```

你也可以把一个长字符串强制转换为零结尾字符串，以下是适用规则：

- 若 S 是一个长串表达式，PChar（S）把 S 转换为零结尾字符串，它返回一个指针，这个指针指向 S 中的第 1 个字符。

  在 Windows 下：比如，若 Str1 和 Str2 是长字符串，你可以这样调用 Win32 API 函数 MessageBox：

  ```
  MessageBox(0, PChar(Str1), PChar(Str2), MB_OK);
  ```

  在 Linux 下：比如，若 Str 是一个长字符串，你可以这样调用系统函数 opendir：

  ```
  opendir(PChar(Str));
  ```

- 你也能使用 Pointer（S）把一个长字符串转换为无类型指针，当 S 为空时，转换返回 **nil**。
- 当把一个长串**变量**转换为指针时，直到变量被赋一个新值或超出范围，否则指针都将是有效的。而转换其它长串**表达式**时，只有在发生转换的语句中，这个指针才是有效的。
- 当转换一个长串表达式为指针时，通常要把指针作为只读的。只有满足所有以下条件时，你才能安全地使用指针修改长字符串：
  - ✓ 表达式是一个长串变量；
  - ✓ 字符串非空；
  - ✓ 字符串是唯一的，也就是引用计数为 1。要保证字符串是唯一的，调用 SetLength、SetString 或者 UniqueString 过程。
  - ✓ 转换发生后，字符串没有被修改过；

✓ 被修改的字符都在字符串中。请小心不要对指针使用（索引）边界检查。

当混合使用 WideString 和 PWideChar 时，这些规则同样适用。

Structured types（结构类型）

Structured types: Overview（概述）

结构类型的一个实例可包含多个值。结构类型包括集合、数组、记录，也包括类、类引用（class-reference）和接口类型。除了集合只能包含有序值以外，结构类型可以包含其它的结构类型，且结构的层次不受限制。

默认情况下，一个结构类型的值被圆整为字（word）或者双字（double-word），这样访问起来更迅速。当声明一个结构类型时，可以包含关键字 **packed**，这将对数据的存储进行压缩（并不是压缩，只是不再圆整数据，而保留它的自然结构）。比如：

```
type TNumbers = packed array[1..100] of Real;
```

使用 **packed** 使数据访问变慢，并且在使用字符数组的情况下，能影响类型兼容性。

Sets（集合）

集合是同一种有序类型的值的聚集，它们包含的值没有内在的顺序，且一个值在集合中包含两次并没有实际意义。

一个集合类型的取值范围，是构成它的有序类型（称为基础类型）的幂，也就是说，集合可能的值是基础类型的所有子集，也包含空集。基础类型可能的值不要超过 256 个，并且它们的序数必须在 0 到 255 之间。任何像下面的形式：

```
set of baseType
```

声明一个集合类型，这里，*baseType* 是一个合适的有序类型。

因为基础类型的值是有限的，因此，集合类型通常使用子界类型来定义。比如下面的声明：

```
type
  TSomeInts = 1..250;
  TIntSet = set of TSomeInts;
```

它声明一个叫做 TIntSet 的集合类型，它的值是从 1 到 250 之间所有可能的选择。你也可以使用下面的语句达到同样的目的：

```
type TIntSet = set of 1..250;
```

有了上面的声明，你就可以像下面这样构造集合了：

```
var Set1, Set2: TIntSet;
 …
Set1 := [1, 3, 5, 7, 9];
Set2 := [2, 4, 6, 8, 10]
```

你也可以直接使用 **set of …**构造直接声明变量：

```
var MySet: set of 'a'..'z';
 …
MySet := ['a','b','c'];
```

其它集合类型的实例包括：

```
set of Byte
set of (Club, Diamond, Heart, Spade)
```

```
set of Char;
```

运算符 **in** 判断集合的成员关系：

```
if 'a' in MySet then ... { do something } ;
```

每个集合类型可包含空集，用**[]**来表示。

Arrays（数组）

Arrays: Overview（概述）

一个数组是由相同类型的（称为基础类型）、经过索引的元素组成的聚集。因为每个元素有唯一的索引，所以，数组和集合不同，它可以包含多个相同的值。数组可以静态分配内存，也可以动态分配。

Static arrays（静态数组）

静态数组类型以下面的格式声明：

```
array[indexType1, ..., indexTypen] of baseType
```

这里，每个 *indexType* 是有序类型并且范围不超过 2G。因为 *indexType* 是数组的索引，所以，数组包含的元素个数由 *indexType* 的范围限定。在实际应用中，*indexType* 通常是整数子界类型。

最简单的是一维数组，它只有一个 *indexType*，比如：

```
var MyArray: array[1..100] of Char;
```

声明了一个变量 MyArray，它是一个有 100 个字符的数组。给定上面的声明，MyArray[3]表示数组中的第 3 个字符。若声明了一个静态数组，虽然并没有给每一个元素赋值，但未用的元素仍分配内存并包含一个随机值，这和未初始化的变量类似。

A multidimensional array is an array of arrays. For example,

一个多维数组是数组的数组，比如：

```
type TMatrix = array[1..10] of array[1..50] of Real;
```

就等价于

```
type TMatrix = array[1..10, 1..50] of Real;
```

不论用哪种方式声明，它表示一个有 500 个实数值的数组。一个 TMatrix 类型的变量 MyMatrix，可使用这样的索引：MyMatrix[2,45]，或像这样：MyMatrix[2][45]。同样，

```
packed array[Boolean,1..10,TShoeSize] of Integer;
```

就等价于

```
packed array[Boolean] of packed array[1..10] of packed array[TShoeSize] of
Integer;
```

标准函数 Low 和 High 作用于数组类型（的标志符）或变量，它们返回数组第 1 个索引（类型）的最小值和最大值；Length 返回数组第 1 维的元素个数。

一维、

一维、压缩的（packed）、Char 类型的静态数组称为 packed string，它和字符串类型兼容，也和其它具有相同元素个数的 packed string 兼容。请参考 Type compatibility and identity。

array[0..*x*] **of** Char 类型的数组，是 0 下标开始的字符数组，它用来存储零结尾字符串，并且和 PChar 类型兼容。参考 Working with null-terminated strings。

Dynamic arrays（动态数组）

动态数组没有固定大小和长度，相反，当你给它赋值或把它传给 SetLength 函数时，它的内存被重新分配。动态数组以下面的形式声明：

```
array of baseType
```

比如

```
var MyFlexibleArray: array of Real;
```

声明一个实数类型的一维动态数组。声明并没有为 MyFlexibleArray 分配内存，要在内存中创建数组，要调用 SetLength。比如，以上面的声明为例：

```
SetLength(MyFlexibleArray, 20);
```

分配一个由 20 个实数构成的数组，索引号从 0 到 19。动态数组的索引总是整数，并从 0 开始。动态数组变量实际是指针，并和长字符串一样使用引用计数进行管理。要取消动态数组的分配，给它的变量赋值 **nil**，或者把变量传给 Finalize。在没有其它引用的情况下，这两种方法都将消除数组。0 长度动态数组的值为 **nil**。不要对一个动态数组变量使用运算符‘^’，也不要对它使用 New 或 Dispose 过程。
若 X 和 Y 是同一类型的动态数组变量，X := Y 使 X 指向和 Y 相同的数组（在这个操作之前，不必给 X 分配内存）。不像字符串和静态数组，动态数组不会在被写之前自动拷贝。比如，在下面的代码执行后

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  A[0] := 1;
  B := A;
  B[0] := 2;
end;
```

A[0]的值是 2（若 A 和 B 是静态数组，A[0]仍然是 1）。
使用索引给动态数组赋值（比如，MyFlexibleArray[2] := 7），不会为数组重新分配内存；编译时，索引边界检查也不会给出提示。
当比较动态数组变量时，是比较它们的引用（这里的值是一个地址），而不是它们的值。所以，下面的代码执行后

```
var
  A, B: array of Integer;
begin
  SetLength(A, 1);
  SetLength(B, 1);
  A[0] := 2;
  B[0] := 2;
end;
```

A = B 返回 False，但 A[0] = B[0]返回 True。
要截断一个动态数组，把它传给 SetLength 或 Copy，并把返回的值赋给数组变量（SetLength 通常更快）。比如，若 A 是一个动态数组，执行 A := SetLength(A, 0, 20)，除 A 的前 20 个元素外，其它都将被截取掉。
一旦一个动态数组被分配内存，你可以把它传给几个标准函数：Length、High 和 Low。Length 返回数组

的元素个数，High 返回最大数组的最大索引（也就是 Length−1），Low 返回 0。对于长度为 0 的数组，High 返回−1（得到反常结果 *High* < *Low*）。

注意：有些函数或过程在声明时，数组参数表示为 array of *baseType*，没有指明索引类型。比如，

```
function CheckStrings(A: array of string): Boolean;
```

这表明，函数可用于（指定的）基础类型的所有数组，而不管它们的大小和索引，也不管它们是静态分配还是动态分配。请参考 Open array parameters。

Multidimensional dynamic arrays（多维动态数组）

要声明多维动态数组，使用复合 **array of ...** 结构，比如，

```
type TMessageGrid = array of array of string;
var Msgs: TMessageGrid;
```

声明一个二维字符串数组。要实例化这个数组，应用两个整数参数调用 SetLength。比如，若 I 和 J 是整数变量，

```
SetLength(Msgs,I,J);
```

给它分配内存，Msgs[0,0]表示它的一个元素。

你也能创建不规则的多维动态数组。第一步是调用 SetLength，给它传递参数作为前面的（几个）索引。比如，

```
var Ints: array of array of Integer;
SetLength(Ints,10);
```

为 Ints 分配了 10 行，但没有分配列。接下来，你能每次分配一个列（给它们指定不同的长度），比如，

```
SetLength(Ints[2], 5);
```

使 Ints 的第 3 行有 5 个元素。此时（即使其它列没有被分配），你能给它的第 3 行赋值，比如，Ints[2,4] := 6。

下面的例子使用动态数组（IntToStr 函数在 SysUtils 单元声明）来创建一个字符串的三角形矩阵。

```
var
  A : array of array of string;
  I, J : Integer;
begin
  SetLength(A, 10);
  for I := Low(A) to High(A) do
  begin
    SetLength(A[I], I);
    for J := Low(A[I]) to High(A[I]) do
      A[I,J] := IntToStr(I) + ',' + IntToStr(J) + ' ';
  end;
end;
```

Array types and assignments（数组类型和赋值）

Arrays are assignment-compatible only if they are of the same type. Because Pascal uses name-equivalence for types, the following code will not compile.

只有数组是相同类型时，它们才是赋值兼容的。因为 Pascal 使用‘名称’代表‘类型’，所以下面的代码无法编译：

```
var
  Int1: array[1..10] of Integer;
  Int2: array[1..10] of Integer;
  ...
Int1 := Int2;
```

要使赋值能够工作，要如下声明变量

```
var Int1, Int2: array[1..10] of Integer;
```

或

```
type IntArray = array[1..10] of Integer;
var
  Int1: IntArray;
  Int2: IntArray;
```

Records（记录）

About records（关于记录）

记录（类似于其它语言中的结构）表示不同种类的元素的集合，每个元素称为“字段”，声明记录类型时要为每个字段指定名称和类型。声明记录的语法是

```
type recordTypeName = record
  fieldList1: type1;
  ...
  fieldListn: typen;
end
```

这里，*recordTypeName* 是一个有效标志符，每个 *type* 表示一种类型，每个 *fieldList* 是一个有效标志符或用逗号隔开的标志符序列，最后的分号是可选的。（哪个分号？是最后一个字段的，还是 end 后面的？）

比如，下面的语句声明了一个记录类型 TDateRec：

```
type
  TDateRec = record
    Year: Integer;
    Month: (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
    Day: 1..31;
  end;
```

TDateRec 包含 3 个字段：一个整数类型的 Year，一个枚举类型的 Month，和另一个子界类型的 Day。标志符 Year、Month 和 Day 是 TDateRec 的字段，它们的行为就像变量。声明并不会为 Year、Month 和 Day 分配内存，只有在实例化时才进行分配，像下面的样子：

```
var Record1, Record2: TDateRec;
```

上面的变量声明创建了两个 TDateRec 实例，分别叫做 Record1 和 Record2。

你可以用记录名作限定符、通过字段名来访问字段：

```
Record1.Year := 1904;
Record1.Month := Jun;
Record1.Day := 16;
```

或使用 **with** 语句：

```
with Record1 do
begin
```

```
    Year := 1904;
    Month := Jun;
    Day := 16;
  end;
```

现在，你可以把 Record1 的值拷贝给 Record2：

```
  Record2 := Record1;
```

因为字段名的范围被限定在记录本身，你不必担心字段名和其它变量发生冲突。

Instead of defining record types, you can use the record ... construction directly in variable declarations:

除了定义记录类型，你也可以使用 **record ...**构造直接声明变量：

```
  var S: record
    Name: string;
    Age: Integer;
  end;
```

但是，这样不能让你重复使用类型声明，并且，这样声明的类型不是赋值兼容的，即使它们（记录）的结构完全相同。

Variant parts in records（记录中的变体部分，变体记录）

一个记录类型能拥有变体部分，它看起来就像 case 语句，在声明中，变体部分必须跟在其它字段的后面。要声明一个变体记录，使用下面的语法：

```
  type recordTypeName = record
    fieldList1: type1;
    ...
    fieldListn: typen;
  case tag: ordinalType of
    constantList1: (Variant1);
    ...
    constantListn: (Variantn);
  end;
```

声明的前面部分（直到关键字 **case**）和标准记录类型一样，声明的其余部分（从 **case** 到最后一个可选的分号，）称为变体部分，在变体部分

- *tag* 是可选的，它可以是任何有效标志符。如果省略了 tag，也要省略它后面的冒号（**:**）。
- *ordinalType* 表示一种有序类型。
- 每个 *constantList* 表示一个 *ordinalType* 类型的常量，或者用逗号隔开的常量序列。在所有的常量中，一个值不能出现多次。
- 每个 *Variant* 是一个由逗号隔开的、类似于 *fieldList*: *type* 的声明列表，也就是说，*Variant* 有下面的形式：

  ```
  fieldList1: type1;
      ...
  fieldListn: typen;
  ```

  这里，每个 *fieldList* 是一个有效标志符，或是由逗号隔开的标志符列表，每个 *type* 表示一种类型，最后一个分号是可选的。这些类型不能是长字符串、动态数组、变体类型或接口（都属于动态管理类型），也不能是包含上述类型的结构类型，但它们可以是指向这些类型的指针。

变体记录类型语法复杂，但语义却很简单：记录的变体部分包含几个变体类型，它们共享同一个内存区域。你能在任何时候，对任何一个变体类型的任何字段读取或写入，但是，当你改变了一个变体的一个

字段，又改变了另一个变体的一个字段时，你可能覆盖了自己的数据。如果使用了 *tag*，它就像记录中非变体部分一个额外的字段，它的类型是 *ordinalType*。

变体部分有两个目的。首先，假设你想创建这样一个记录：它的字段有不同类型的数据，但你知道，在一个（记录）实例中你永远不需要所有的字段，比如：

```
type
  TEmployee = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Salaried: Boolean of
    True: (AnnualSalary: Currency);
    False: (HourlyWage: Currency);
end;
```

这里的想法是，每个雇员或者是年薪，或者是小时工资，但不能两者都有。所以，当你创建一个 TEmployee 的实例时，没必要为每个字段都分配内存。在上面的情形中，变体间的唯一区别在于字段名，但更简单的情况是字段拥有不同的类型。看一下更复杂的例子：

```
type
  TPerson = record
  FirstName, LastName: string[40];
  BirthDate: TDate;
  case Citizen: Boolean of
    True: (Birthplace: string[40]);
    False: (Country: string[20];
            EntryPort: string[20];
            EntryDate, ExitDate: TDate);
  end;
type
  TShapeList = (Rectangle, Triangle, Circle, Ellipse, Other);
  TFigure = record
    case TShapeList of
      Rectangle: (Height, Width: Real);
      Triangle: (Side1, Side2, Angle: Real);
      Circle: (Radius: Real);
      Ellipse, Other: ();
  end;
```

对每个记录类型的实例，编译器分配足够的内存以容纳最大变体类型的所有字段。可选的 *tag* 和 *constantLists*（像上面例子中的 Rectangle、Triangle 等）对于编译器管理字段没有任何作用，它们只是为了程序员的方便。

使用变体记录的第二个原因是，你可以把同一个数据当作不同的类型进行处理，即使在编译器不允许类型转换的场合。比如，在一个变体类型中，它的第一个字段是 64 位实数，在另一个变体类型中，第一个字段是 32 位整数，你可以把一个值赋给实数（字段），然后再当作整数来读取它的前 32 位值（比如，把它传给一个需要整数参数的函数）。

File types（文件类型）

file 是由相同类型的元素组成的有序集合。标准 I/O 例程使用内置（预定义）的 TextFile 或 Text 类型，它们表示一个包含字符的文件，这些字符是以行的形式进行组织的。想了解更多关于文件输入和输出的信

息，请参考 Standard routines and I/O（标准例程和 I/O）。
要声明一个文件类型，使用下面的语法：

```
type fileTypeName = file of type
```

这里，*fileTypeName* 是任何有效的标志符，*type* 是一个固定大小的类型。指针类型（不管是隐含的还是直接的）是不可以的，所以，文件不能包含动态数组、长字符串、类、对象、指针、变体类型、其它文件或包含以上类型的结构类型。
比如，

```
type
  PhoneEntry = record
    FirstName, LastName: string[20];
    PhoneNumber: string[15];
    Listed: Boolean;
  end;
  PhoneList = file of PhoneEntry;
```

声明了一个记录姓名和电话号码的文件类型。
在声明变量时，你也可以直接使用 **file of ...**结构，比如，

```
var List1: file of PhoneEntry;
```

单独的一个 **file** 表示一个无类型文件：

```
var DataFile: file;
```

要了解更多信息，请参考 Untyped files（无类型文件）。
数组和记录中不能包含文件类型。

Pointers and pointer types（指针和指针类型）

Pointers and pointer types（指针和指针类型）

指针是一个表示内存地址的变量。当一个指针包含另一个变量的地址时，我们认为它指向这个变量在内存中的位置，或指向数据存储的地方。对于数组或其它结构类型，指针指向的是结构中第一个元素的地址。
指针被*类型化*以表示在它指定的位置上存储某一类型的数据。Pointer 类型表示一个任意类型的指针，而指定类型的指针只表示特定类型的数据。指针在内存中占据 4 个字节。

Overview of pointers（指针概述）

要了解指针如何工作，看下面的例子：

```
1    var
2      X, Y: Integer;      // X 和 Y 是整数变量
3      P: ^Integer;        // P 指向一个整数
4    begin
5      X := 17;            // 给 X 赋值
6      P := @X;            // 把 X 的地址赋给 P
7      Y := P^;            // dereference P；把结果赋给 Y
8    end;
```

第 2 行声明 X 和 Y 为整数类型的变量，第 3 行声明 P 是一个指向整数的指针，这表明 P 可以指向 X 或

Y 的存储位置。第 5 行把一个值赋给 X，第 6 行把 X 的地址（用@X 表示）赋给 P。最后，在第 7 行，取得 P 所指位置的值（用 P^表示）并把它赋给 Y。这些代码执行后，X 和 Y 有相同的值，即 17。

@运算符，这里我们用来取得一个变量的地址，它同样可用于函数或过程。要了解更多信息，请参考 The @ operator 和 Procedural types in statements and expressions。

^符号有两个用途，在我们的例子中都用到了。当它出现在一个类型标志符前面时：

```
^typeName
```

它表示一个指向 *typeName* 类型的变量的指针；当它出现在一个指针变量的后面时：

```
pointer^
```

它表示对指针解除引用，换句话说，它返回在指针所指向的地址处保存的值。

我们的例子看起来是在兜圈子，它只不过是把一个变量的值复制给另一个变量而已，我们完全可以通过一个简单的赋值语句来完成，但指针有几个用途：首先，理解指针能帮助你理解 Object Pascal，因为经常在代码中虽然没有明确使用指针，但它们却在背地里发挥作用。使用大的、动态分配内存（块）的任何数据类型都使用指针。例如，长字符串就是一个隐含的指针，类变量也是；此外，一些高级的编程技术需要使用指针。

最后，指针有时是跳过 Object Pascal 严格的（数据）类型匹配的唯一方法。使用一个通用指针（Pointer）来引用一个变量，并把它转换为其它类型，然后重新引用它，这样你就可以把它作为任何类型对待。比如，下面的代码把一个实数变量的值赋给一个整数变量。

```
type
  PInteger = ^Integer;
var
  R: Single;
  I: Integer;
  P: Pointer;
  PI: PInteger;
begin
  ...
  P := @R;
  PI := PInteger(P);
  I := PI^;
end;
```

当然，实数和整数有不同的存储格式，上面的赋值只简单地把 R 的二进制数据赋给 I，并不是实际转换。

除了使用@运算符，你也可以使用几个标准例程给一个指针赋值。New 和 GetMem 过程把一个内存地址赋给指针，而 Addr 和 Ptr 函数则返回一个指向特定变量或地址的指针。

像 P1^.Data^表示的那样，对指针解除引用可用作限定符，也可以被限定。

保留字 nil 是一个特殊常量，可赋给任何指针（类型）。当 nil 被赋给一个指针时，指针不表示任何东西。

Pointer types（指针类型）

About pointer types（关于指针类型）

使用下面的语法，你能声明一个任意类型的指针，

```
type pointerTypeName = ^type
```

当定义一个记录类型（或其它数据类型）时，习惯上也就定义了一个此类型的指针，这使得处理更容易，我们不需要拷贝一大块内存。

标准指针类型有许多理由存在，最通用的是 Pointer，它可以指向任何数据类型，但不能对它解除引用，

在 Pointer 类型变量的后面使用^运算符会引发编译错误。要访问一个 Pointer 类型引用的变量，首先把它转换为其它指针类型，然后再解除引用。

Character pointers（字符指针）

基本（fundamental）类型 PAnsiChar 和 PWideChar 分别表示 AnsiChar 和 WideChar 值的指针，一般（generic）类型 PChar 表示一个指向 Char 的指针（在当前实现中，它表示 AnsiChar）。这些字符指针用来操纵零结尾字符串（参考 Working with null-terminated strings）。

Other standard pointer types（其它标准指针类型）

System 和 SysUtils 单元定义了许多常用的标准指针类型：

Pointer type	Points to variables of type
PAnsiString，PString	AnsiString
PByteArray	TByteArray (declared in SysUtils). Used to typecast dynamically allocated memory for array access.
PCurrency，PDouble，PExtended，PSingle	Currency，Double，Extended，Single
PInteger	Integer
POleVariant	OleVariant
PShortString	ShortString. Useful when porting legacy code that uses the old PString type.
PTextBuf	TTextBuf (declared in SysUtils). TTextBuf is the internal buffer type in a TTextRec file record.)
PVarRec	TVarRec (declared in System)
PVariant	Variant
PWideString	WideString
PWordArray	TWordArray (declared in SysUtils). Used to typecast dynamically allocated memory for arrays of 2-byte values.

Procedural types（过程类型）

Procedural types: Overview（概述）

过程类型允许你把过程和函数作为“值”看待，它可以赋给变量或传给其它过程和函数。比如，假设你定义了一个叫做 Calc 的函数，它有两个整型参数并返回一个整数值：

```
function Calc(X,Y: Integer): Integer;
```

你可以把 Calc 函数赋给变量 F：

```
var F: function(X,Y: Integer): Integer;
F := Calc;
```

我们只取过程或函数头（heading）并把 **procedure** 或 **function** 后面的标志符去掉，剩下的就是过程类型的名称。你可以在声明变量时直接使用这样的名称（就像上面的例子一样），也可以声明新类型：

```
type
  TIntegerFunction = function: Integer;
```

```
    TProcedure = procedure;
    TStrProc = procedure(const S: string);
    TMathFunc = function(X: Double): Double;
  var
    F: TIntegerFunction;      {F 是一个无参数、返回整数值的函数}
    Proc: TProcedure;         { Proc 是一个无参数过程}
    SP: TStrProc;             { SP 是一个使用 string 类型参数的过程}
    M: TMathFunc;             { M 是一个使用 Double 类型参数、返回 Double 值的函数}

  procedure FuncProc(P: TIntegerFunction);   { FuncProc 是一个过程，它的参数是
                                               一个无参数、返回整数值的函数}
```

上面的所有变量都是**过程指针**，也就是指向过程或函数地址的指针。若想引用一个实例对象的方法（参考 Classes and objects），你需要在过程类型的名称后面加上 **of object**。比如

```
  type
    TMethod = procedure of object;
    TNotifyEvent = procedure(Sender: TObject) of object;
```

这些类型表示**方法指针**。方法指针实际上是一对指针：第一个存储方法的地址，第二个存储方法所属的对象的引用。给出下面的声明

```
  type
    TNotifyEvent = procedure(Sender: TObject) of object;
    TMainForm = class(TForm)
      procedure ButtonClick(Sender: TObject);
      ...
    end;
  var
    MainForm: TMainForm;
    OnClick: TNotifyEvent
```

我们就可以进行下面的赋值：

```
  OnClick := MainForm.ButtonClick;
```

两个过程类型是兼容的，如果它们具有

- 相同的调用约定，
- 相同类型的返回值（或没有返回值），并且具有
- 相同数目的参数，并且相应位置上的类型也相同（参数名无关紧要）

过程指针和方法指针是不兼容的。**nil** 可以赋给任何过程类型。

嵌套的过程和函数（在其它例程中声明的例程）不能被用作过程类型值，内置的过程和函数也不可以。若想使用内置的过程作为过程类型值，比如 Length，你可以给它加一个包装：

```
  function FLength(S: string): Integer;
  begin
    Result := Length(S);
  end;
```

Procedural types in statements and expressions（语句和表达式中的过程类型）

当一个过程变量出现在赋值语句的左边时，编译器期望右边是一个过程类型值。赋值操作把左边的变量

当作指针，它指向右边指示的过程或函数。但在其它情形，使用过程变量将调用它引用的过程或函数，你甚至可以对过程变量传递参数：

```
var
  F: function(X: Integer): Integer;
  I: Integer;
function SomeFunction(X: Integer): Integer;
 ...
F := SomeFunction;  // 把 SomeFunction 赋给 F
I := F(4);          // 调用函数，把结果赋给 I
```

在赋值语句中，左边变量的类型决定右边的过程（或方法）指针的解释，比如

```
var
  F, G: function: Integer;
  I: Integer;
function SomeFunction: Integer;
 ...
F := SomeFunction;  // 把 SomeFunction 赋给 F
G := F;             // 拷贝 F 到 G
I := G;             // 调用函数，把结果赋给 I
```

第 1 个赋值语句把一个过程类型值赋给 F，第 2 个语句把这个值拷贝给另一个变量，第 3 个语句调用引用的函数并把结果赋给 I。因为 I 是一个整数变量，而不是过程类型，所以最后的赋值实际上是调用函数（它返回一个整数值）。

在一些情况下，如何解释过程变量并不是很清楚，看下面的语句

```
if F = MyFunction then ...;
```

在此情况下，F 导致函数调用：编译器调用 F 指向的函数，然后调用函数 MyFunction，然后比较结果。规则是，只要过程变量出现在表达式中，它就表示是调用引用的过程或函数。在上面的例子中，若 F 引用一个过程（没有返回值），或 F 引用的函数需要参数，则引发编译错误。要比较 F 和 MyFunction 的过程值，使用

```
if @F = @MyFunction then ...;
```

@F 把 F 转换为无类型指针变量，它包含的是地址，@MyFunction 返回的是 MyFunction 的地址。

要取得过程变量的内存地址（而不是它包含的地址），使用**@@**。比如，@@F 返回 F 的地址。

@运算符也可以用来把一个无类型指针值赋给过程变量，比如

```
var StrComp: function(Str1, Str2: PChar): Integer;
 ...
@StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');
```

调用 GetProcAddress 函数，并使 StrComp 指向结果。

过程变量可以是 **nil** 值，但此时调用它会引发错误。要测试一个过程变量是否被赋值，使用标准函数 Assigned：

```
if Assigned(OnClick) then OnClick(X);
```

Variant types（变体类型）

Variant types: Overview（概述）

有时，我们需要使用这样的数据：它们的类型是可变的，或者在编译时它们的类型不能确定。在这种情况下，一个选择是使用变体类型作为变量或参数，它们可以在运行时改变类型。变体类型提供了更大的

灵活性，但却比普通变量需要更多的内存，并且操作起来更慢。并且，对变体类型的非法使用会导致运行时错误，而普通变量的此类错误能在编译时被发现。你也可以创建自定义变体类型。

默认情况下，除记录、集合、静态数组、文件、类、类引用和指针外，变体类型能存储任何其它类型的值。换句话说，除结构类型和指针外，变体类型能存储其它的任何类型；变体类型能存储接口，并能通过它使用接口的方法和属性（参考 Object interfaces）；变体类型能存储动态数组，也能存储一种特殊的静态数组：变体数组（Variant array）。变体类型能和其它变体类型、整数、实数、字符串和布尔值在表达式和赋值语句中混合使用，编译器自动完成类型转换。

包含字符串的变体类型不能使用索引，也就是说，若 V 是一个存储字符串的变体类型，则 V[1]会导致运行时错误。

你可以通过自定义来扩展变体类型，从而能存储任意值。比如，你可以定义一个使用索引的变体字符串类型，或者让它存储特定的类引用、记录或静态数组。自定义变体类型通过 TCustomVariantTyp 的子类来创建。

变体类型占用 16 字节的内存，包含一个类型码和一个值（或指向这个值的指针），值的类型由类型码指定。所有的变体类型在创建时被初始化为 Unassigned，Null 表示未知或没有数据。

标准函数 VarType 返回变体类型的类型码，常量 varTypeMask 是一个位掩码，用来从 VarType 的返回值中提取类型码，所以，在下面的例子中

```
VarType(V) and varTypeMask = varDouble
```

若 V 包含 Double 或 Double 数组，则它返回 True（掩码简单地隐藏了第一位，它指示变体类型是否存储一个数组）。在 System 单元定义的 TVarData 记录类型能被用来转换变体类型，并且可以访问它们的内部构造。请参考在线帮助取得 VarType 的类型码列表，要注意，在 Object Pascal 的未来实现中，可能会添加新的类型码。

Variant types conversions（Variant 类型转换）

所有的整数、实数、字符串、字符和布尔类型与 Variant 是赋值兼容的。表达式能明确转换为 Variant，VarAsType 和 VarCast 标准例程能用来改变一个 Variant 的内部表示。下面代码演示了 Variant 的使用以及当 Variant 和其它类型混用时会发生的一些自动转换。

```
var
  V1, V2, V3, V4, V5: Variant;
  I: Integer;
  D: Double;
  S: string;
begin
  V1 := 1;                    { 整数值 }
  V2 := 1234.5678;            { 实数值 }
  V3 := 'Hello world!';       { 字符串 }
  V4 := '1000';               { 字符串 }
  V5 := V1 + V2 + V4;         { 实数值 2235.5678}
  I := V1;                    { I = 1 (整数值) }
  D := V2;                    { D = 1234.5678 (实数值) }
  S := V3;                    { S = 'Hello world!' (字符串) }
  I := V4;                    { I = 1000 (整数值) }
  S := V5;                    { S = '2235.5678' (字符串) }
end;
```

编译器依据下面的规则进行类型转换：

Source/Target	integer	real	string	character	Boolean
integer	转换为整数	转换为实数	转换为字符串形式	同 string（左边）	若 0 则返回 False，其它为 True。
real	圆整为最接近的整数	转换为实数	按区域设置转换为字符串形式	同 string（左边）	若 0 则返回 False，其它为 True。
string	转换为整数，需要的话会截断；若不是数字则引发异常。	按区域设置转换为实数，若不是数字则引发异常。	转换为 string/character 格式	同 string（左边）	若是字符串“false”（大小写无关）或值为 0 的数字串，则返回 False，若是字符串“true”或值为非 0 的数字串，则返回 True。其它情况产生异常。
character	同 string（上边）	同 string（上边）	同 string（上边）	和 string 转换为 string 相同	同 string（上边）
Boolean	False=0，True=-1（若是 Byte，则是 255）	False=0，True＝-1	False=“0”，True＝“-1”	同 string（左边）	False=False，True＝True
Unassigned	0	0	空串	同 string（左边）	False
Null	引发异常	引发异常	引发异常	同 string（左边）	引发异常

超出范围的赋值经常使目标变量取得它的最大值，无效的赋值或转换引发 EVariantError 异常。
System 单元声明的 TDateTime 实数类型有特殊的转换规则，当 TDateTime 转换为任何其它类型时，它被看作 Double 类型；当整数、实数或布尔类型转换为 TDateTime 时，它先转换为 Double，然后当作 date-time 值来读取；当一个字符串被转换为 TDateTime 时，它使用区域设置解释为 date-time 值；当 **Unassigned** 值转换为 TDateTime 时，它被看作实数或整数的 0；把 **Null** 值转换为 TDateTime 会引发异常。
在 Windows 下，若 Variant 引用一个 COM 接口，任何对它的转换尝试都会读取它的默认属性并把它的值转换为需要的类型。若对象没有默认属性，会引发异常。

Variants in expressions（表达式中的 Variant）

除了^、**is** 和 **in**，所有运算符都可以使用 Variant 作为运算数。对 Variant 的操作返回 Variant 值；若有一个运算数是 **Null** 则结果为 **Null**；若有一个运算数为 **Unassigned** 则引发异常。在二元运算中，若只有一个运算数是 Variant，则另一个被转换为 Variant。
运算结果的类型由它的运算数决定。通常，适用于静态（边界）类型的规则对 Variant 也是有效的。比如，若 V1 和 V2 是 Variant 类型，分别存储整数和实数，那么 V1 + V2 返回一个有实数值的 Variant。但是，使用 Variant，你可以在二元运算中组合两个对于静态类型是不允许的值。可能的情况下，编译器使用在 Variant type conversion 中总结的规则转换不匹配的 Variant，比如，若 V3 和 V4（都是 Variant 类型）分别存储一个数字串和一个整数，表达式 V3 + V4 返回一个整数值 Variant，在运算前，数字串被转换为整数。

Variant arrays（Variant 数组）

你不能把一个普通的静态数组赋给 Variant，取而代之的是，通过调用 VarArrayCreate 或 VarArrayOf 两者之一来创建 Variant 数组。比如，

```
V: Variant;
 ...
V := VarArrayCreate([0,9], varInteger);
```

它创建一个整数类型的 Variant 数组（长度为 10）并赋给 Variant V。数组被索引为 V[0]、V[1]，等等，但不能把数组的一个元素作为 **var** 参数。Variant 数组总是使用整数作为索引。
VarArrayCreate 中的第二个参数是数组基础类型的类型码，要查看这些类型码列表，参考 VarType。永远不要向 VarArrayCreate 传递 varString，要创建字符串类型的 Variant 数组，使用 varOleStr。
Variant 能存储有不同大小、维数和基础类型的 Variant 数组。Variant 数组的元素可以是除了 ShortString 和 AnsiString 以外 Variant 允许的任何类型，并且，若数组的基础类型是 Variant，它的元素甚至可以具有不同的类型。使用 VarArrayRedim 函数来更改 Variant 数组的大小。其它用于 Variant 数组的标准例程包括
VarArrayDimCount 、 VarArrayLowBound 、 VarArrayHighBound 、 VarArrayRef 、 VarArrayLock 和 VarArrayUnlock.
当一个包含 Variant 数组的 Variant 被赋给另一个 Variant 或作为值参传递时，整个数组被拷贝。除非必须，否则不要使用这样的操作，因为对内存操作很低效。

OleVariant（OleVariant）

OleVariant 类型在 Windows 和 Linux 平台都存在。Variant 和 OleVariant 的主要区别是，Variant 能包含只有当前程序才能理解的数据类型，OleVariant 只包含为 Ole 自动化兼容而定义的数据类型，它说明，这些数据类型能在程序间或通过网络传送，而不必担心另一端是否知道如何处理它们。
当把一个包含自定义数据（比如 Pascal 字符串，或新定义的 Variant 类型中的一种）的 Variant 赋给 OleVariant 时，运行库尝试把 Variant 转换为 OleVariant 标准数据类型中的一种（比如 Pascal 字符串转换为 OleBSTR 字符串）。比如，若一个包含 AnsiString 的 Variant 被赋给 OleVariant，AnsiString 变为 WideString。当把 Variant 作为 OleVariant 参数传递时，前面的转换也是成立的。

Type compatibility and identity（类型兼容性和一致性）

Type compatibility and identity: Overview（概述）

要理解何种表达式能进行何种操作，我么需要区分几种类型和值的兼容性，它们包括类型一致性、类型兼容性和赋值兼容性。

Type identity（类型一致性）

Type identity is almost straightforward. When one type identifier is declared using another type identifier, without qualification, they denote the same type. Thus, given the declarations
类型一致性非常直接。当一种类型（标志符）使用另一种类型声明时（没有限制），它们表示相同的类型。这样，下面的声明

```
type
  T1 = Integer;
  T2 = T1;
  T3 = Integer;
  T4 = T2;
```

T1、T2、T3、T4 和 Integer 都是指的同一种类型。要声明一种不同的类型，在声明中重复 type 关键字。比如

```
type TMyInteger = type Integer;
```

创建了一种新类型 TmyInteger，它和 Integer 不同。

起类型声明作用的语句构造，它们每次出现都声明一种不同的类型，所以下面的声明

```
type
  TS1 = set of Char;
  TS2 = set of Char;
```

创建了两种不同的类型：TS1 和 TS2。类似的，变量声明

```
var
  S1: string[10];
  S2: string[10];
```

创建两个不同的变量。要创建相同类型的变量，使用

```
var S1, S2: string[10];
```

或

```
type MyString = string[10];
var
  S1: MyString;
  S2: MyString;
```

Type compatibility（类型兼容性）

每个类型和自己兼容。如果至少满足下面的一个条件，两个不同的类型也是兼容的：

- 它们都是实数类型
- 它们都是整数类型
- 一种类型是另一种的子界类型
- 两个都是同一种类型的子界类型
- 两个都是集合类型，并且它们的基础类型是兼容的
- 两个都是 packed-string 类型，并且有相同数目的元素
- 一个是字符串类型，另一个是字符串、packed-string 和 Char 类型
- 一个是 Variant 类型，另一个是整数、实数、字符串、字符或布尔类型
- 两个都是类、类引用或接口类型，并且一个来自于（继承）另一个。
- 一个是 PChar 或 PWideChar，另一个是 0 下标开始的字符数组（array[0..n] of Char）
- 一个是 Pointer（无类型指针）类型，另一个是任何指针类型
- 两个是同一种类型的指针，并且开启了编译器指示字**{$T+}**
- 两个都是过程类型，它们有相同的返回类型，并且参数的个数、位置和类型都相同。

Assignment-compatibility（赋值兼容性）

赋值兼容性不是一种对称关系。T1 是一个变量，T2 是一个表达式，若 T2 的值在 T1 的取值范围内，并

且至少下面一个条件成立，则 T2 可以赋给 T1：

- T1 和 T2 是同一种类型，并且不是文件类型或包含文件类型的结构类型
- T1 和 T2 是兼容的有序类型
- T1 和 T2 都是实数类型
- T1 是实数类型，T2 是整数类型
- T1 是 PChar 类型或任何字符串类型，而 T2 是字符串常量
- T1 和 T2 都是字符串类型
- T1 是字符串类型，T2 是字符或 packed-string 类型
- T1 是一个长串类型，T2 是一个 PChar 类型
- T1 和 T2 是兼容的 packed-string 类型
- T1 和 T2 是兼容的集合类型
- T1 和 T2 是兼容的指针类型
- T1 和 T2 都是类、类引用或接口类型，并且 T2 继承自 T1
- T1 是一个接口类型，T2 是实现 T1 的一个类
- T1 是 PChar 或 PWideChar，T2 是一个 0 下标开始的字符数组（array[0..n] of Char）
- T1 和 T2 是兼容的过程类型（在一些赋值语句中，一个函数或过程的标志符被认为是过程类型）
- T1 是 Variant 类型，T2 是整数、实数、字符串、字符、布尔或接口类型
- T1 是整数、实数、字符串、字符或布尔类型，T2 是 Variant
- T1 是 IUnknown 或 IDispatch 接口类型，T2 是 Variant 类型（若 T1 是 Iunknown，T2 的类型编码必须是 varEmpty、varUnknown 或 varDispatch；若 T1 是 Idispatch，T2 的类型编码必须是 varEmpty 或 varDispatch。）

Declaring types（声明类型）

Declaring types（声明类型）

一个类型声明指定一个标志符，来表示一种数据类型。类型声明的语法为

```
type newTypeName = type
```

这里，*newTypeName* 是一个有效的标志符。比如，给定如下的类型声明

```
type TMyString = string;
```

你就可以声明变量

```
var S: TMyString;
```

一个类型标志符的范围不能包含类型声明本身（指针类型除外），所以举例来说，你不能在声明一个记录时循环使用它。

当声明一个和已有类型完全相同的类型时，编译器把它看作是已有类型的别名。这样，在下面的声明中

```
type TValue = Real;
var
  X: Real;
  Y: TValue;
```

X 和 Y 属于相同的类型。在运行时，没有办法区分 TValue 和 Real 类型。这通常有一些意义，但如果你定义新类型的目的是利用 RTTI（Runtime Type Information），比如，给某个类型赋一个属性编辑器，区分‘不同名称’和‘不同类型’就变得重要了。这种情况下，你使用语法

```
type newTypeName = type type
```

例如

```
type TValue = type Real;
```

这将强制编译器创建一个不同的新类型 TValue。

Variables（变量）

Variables: Overview（概述）

变量是一个标志符，它的值能在运行时改变。从另一个角度看，变量是内存中某个位置的名称，你能使用这个名称读取或写入内存中的这个位置。变量就像数据的容器，因为它们是有类型的，它们也就告诉了编译器如何解释它们存储的数据。

（以下内容摘自《Delphi 技术手册》
Delphi 中定义的绝大部分变量是 System 和 SysInit 单元中的普通变量。这两个单元的区别是：应用程序加载的每个包都共享 System 单元中的变量而拥有自己 SysInit 单元的副本。若你知道自己在做什么，你可以改变它们的值。但是如果不小心的话，也会给 Delphi 造成很大的破坏。其它变量（Self 和 Result）是内置于编译器的，有着特殊的用途。）

Declaring variables（声明变量）

Declaring variables（声明变量）

声明一个变量的基本语法是

```
var identifierList: type;
```

这里，*identifierList* 是由逗号隔开的有效标志符的列表，*type* 是任何有效类型。比如，

```
var I: Integer;
```

声明一个整数类型的变量 I，而

```
var X, Y: Real;
```

声明两个变量 X 和 Y，它们是实数（Real）类型。
连续的变量声明没必要重复使用关键字 **var**：

```
var
  X, Y, Z: Double;
  I, J, K: Integer;
  Digit: 0..9;
  Okay: Boolean;
```

有时，在过程或函数中声明的变量叫做局部变量，而其它变量叫做全局变量。全局变量能在声明时进行初始化，使用语法

```
var identifier: type = constantExpression;
```

这里，*constantExpression* 是任何 *type* 类型的常量表达式。所以下面的声明

```
var I: Integer = 7;
```

相当于声明和语句

```
var I: Integer;
 …
I := 7;
```

同时声明多个变量（比如 **var** X, Y, Z: Real;）时不能包括初始化，**Variant** 和文件类型的变量声明也不能

初始化。
如果你没有明确地初始化一个全局变量，编译器把它初始化为 0。相反，不能在声明局部变量时进行初始化，它们的值是随机的，直到赋给它们一个值。
当你声明变量时，你正在分配内存，当变量不再使用时，这些内存会自动释放。特别是局部变量，它们只存在于程序退出（声明它们的）函数或过程之前。关于变量和内存管理的更多信息，请参考 Memory management。

Absolute address（绝对地址）

你可以创建一个新变量，它和另一个变量在内存的同一个位置。要这样做的话，声明这个新变量时在类型名的后面跟关键字 **absolute**，后面再跟一个已存在（先前声明）的变量。比如，

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

指定变量 StrLen 从 Str 的地址开始。因为短字符串的第一个字节包含字符串的长度，StrLen 的值就是 Str 的长度。
使用 **absolute** 声明时不能初始化变量，也不能组合其它指示字（和 **absolute** 一同使用）。

Dynamic variables（动态变量）

你可以调用 GetMem 或 New 过程来创建动态变量，这种变量在堆中分配内存，它们不能自动管理。一旦你创建了这样一个变量，你要负责在最后释放它的内存。使用 FreeMem 来释放由 GetMem 创建的变量，使用 Dispose 释放由 New 创建的变量。其它能作用于动态变量的标准例程包括 ReallocMem、Initialize、StrAlloc 和 StrDispose。
虽然长字符串、宽字符串、动态数组、Variants 以及接口也是在堆中分配的动态变量，但它们的内存是自动管理的。

Thread-local variables（线程局部变量）

线程局部变量（或线程变量）用于多线程应用程序。线程局部变量类似于全局变量，除了执行的每个线程有自己的这些变量的拷贝，不能从其它线程访问它们。声明线程局部变量时，使用 **threadvar**，而不是 **var**，比如，

```
threadvar X: Integer;
```

线程变量声明

- 不能出现在过程或函数中
- 不能包含初始化
- 不能指定 absolute 指示字

不能创建指针或过程类型的线程变量，也不能在动态调入库中使用线程变量（除了包）。
由编译器管理的动态变量，即长字符串、宽字符串、动态数组、Variants 和接口，能被声明为 **threadvar**，但编译器不能自动释放由每个线程创建的堆内存。若你使用这些类型的线程变量，你要负责释放它们的内存。比如，

```
threadvar S: AnsiString;
S := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
  …
```

```
S := '';          // 释放 S 使用的内存
```

（你能把一个 Variant 设置为 **Unassigned**、把一个接口或动态数组设置为 **nil** 来释放它们。）

Declared constants（声明的常量）

Constants: Overview（概述）

有几种不同的语言构造指明"常量"：有数字常量（也叫做 numerals）和字符串常量（也叫做 character strings 或 string literals），比如'Hello world!'；每个枚举类型也定义常量，表示此类型的一个值；有预定义的常量，比如 **True**、**False** 和 **nil**；最后，还有像变量的常量，它们通过声明来创建。

声明的常量或者是真常量（true constants），或者是类型常量（typed constants）。这两种常量大致类似，但有不同的规则并应用于不同的目的。

True constants（真常量）

True constants（真常量）

真常量是一个声明的标志符，它的值不能更改。比如，

```
const MaxValue = 237;
```

声明一个叫做 MaxValue 的常量，它返回整数值 237。声明一个真常量的语法是

```
const identifier = constantExpression
```

这里，*identifier* 是任何有效标志符，*constantExpression* 是不用执行程序编译器就能计算出的值（参考 Constant expressions 获得更多信息）。

若 *constantExpression* 返回一个有序值，你能使用值转换指定常量的类型。比如

```
const MyNumber = Int64(17);
```

声明一个叫做 MyNumber 的常量，它的类型为 Int64，值为整数 17。否则，声明的常量类型是 *constantExpression* 的类型。

- 若 *constantExpression* 是一个（常量）字符串，声明的常量和任何字符串是类型兼容的；若（常量）字符串的长度为 1，它和任何字符类型也是兼容的。
- 若 *constantExpression* 是实数，它的类型是 Extended。若它是整数，它的类型如下表所示：

常量范围（16 进制）	常量范围（10 进制）	类型
-$8000000000000000..-$80000001	-2^63..-2147483649	Int64
-$80000000..-$8001	-2147483648..-32769	Integer
-$8000..-$81	-32768..-129 Smallint	
-$80..-1	-128..-1	Shortint
0..$7F	0..127	0..127
$80..$FF	128..255	Byte
$0100..$7FFF	256..32767	0..32767
$8000..$FFFF	32768..65535	Word
$10000..$7FFFFFFF	65536..2147483647	0..2147483647
$80000000..$FFFFFFFF	2147483648..4294967295	Cardinal
$100000000..$7FFFFFFFFFFFFFFF	4294967296..2^63-1	Int64

下面是一些声明常量的例子：

```
const
  Min = 0;
  Max = 100;
  Center = (Max - Min) div 2;
  Beta = Chr(225);
  NumChars = Ord('Z') - Ord('A') + 1;
  Message = 'Out of memory';
  ErrStr = ' Error: ' + Message + '. ';
  ErrPos = 80 - Length(ErrStr) div 2;
  Ln10 = 2.302585092994045684;
  Ln10R = 1 / Ln10;
  Numeric = ['0'..'9'];
  Alpha = ['A'..'Z', 'a'..'z'];
  AlphaNum = Alpha + Numeric;
```

Constant expressions（常量表达式）

常量表达式是一个表达式，编译器不用执行程序就能计算出它的值。常量表达式包括：常数（numerals）、字符串常量（character strings）；真常量；枚举类型的值；特殊常量 True、False 和 **nil**；以及由以上类型的元素通过运算符、类型转换和集合构造器构成的表达式。常量表达式不能包括变量、指针或函数调用，除非是调用下面的内置函数：

Abs	High	Low	Pred	Succ
Chr	Length	Odd	Round	Swap
Hi	Lo	Ord	SizeOf	Trunc

常量表达式的定义在 Object Pascal 的语法说明中用到多次。初始化全局变量、定义子界类型、给枚举类型的值赋序数、指定默认参数、书写 case 语句以及定义真常量和类型常量，都需要常量表达式。

下面是常量表达式的例子：

```
100
'A'
256 - 1
(2.5 + 1) / (2.5 - 1)
'Borland' + ' ' + 'Developer'
Chr(32)
Ord('Z') - Ord('A') + 1
```

Resource strings（资源字符串）

资源字符串被当作资源链接到可执行文件或库中，因此，不用重新编译程序就能修改它们。关于更多信息，请参考 Internationalization and localization 以及它相关的主题。

资源字符串的声明像真常量，除了用 **resourcestring** 代替 **const**。表达式等号的右边必须是常量表达式并且返回一个字符串。比如，

```
resourcestring
  CreateError = 'Cannot create file %s';        { for explanations of format specifiers, }
  OpenError = 'Cannot open file %s';            { see 'Format strings' in the online Help}
  LineTooLong = 'Line too long';
```

```
ProductName = 'Borland Rocks\000\000';
SomeResourceString = SomeTrueConstant;
```

编译器自动解决不同库之间的命名冲突问题。

Typed constants（类型常量）

About typed constants（关于类型常量）

不像真常量，类型常量能存储数组、记录、过程和指针类型的值。类型常量不能出现在常量表达式中。在默认的**{$J-}**编译状态下，类型常量不能被赋予新值，实际上，它们是只读变量；但如果使用了**{$J+}**编译器指示字，类型常量能被赋予新值，它们在本质上就像初始化的变量。

像下面这样声明一个类型常量：

```
const identifier: type = value
```

这里，*identifier* 是任何有效标志符，*type* 是除了文件和 **Variant** 类型的任何类型，*value* 是一个 *type* 类型的表达式。比如，

```
const Max: Integer = 100;
```

在大多数情况下，value 必须是一个常量表达式，但如果 type 是数组、记录、过程或指针类型，必须遵循特殊的规则。

Array constants（数组常量）

要声明数组常量，把数组元素的值用括号括起来，值之间用逗号隔开，这些值必须是常量表达式。比如，

```
const Digits: array[0..9] of Char = ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');
```

它声明一个类型常量 Digits，存储一个字符构成的数组。

0 基准字符数组经常用来表示零结尾字符串，因为这个原因，字符串常量能用来初始化字符数组。所以上面的声明可以方便地表示为

```
const Digits: array[0..9] of Char = '0123456789';
```

要定义一个多维数组常量，把每一维的值用括号括起来，之间用逗号隔开。比如，

```
type TCube = array[0..1, 0..1, 0..1] of Integer;
const Maze: TCube = (((0, 1), (2, 3)), ((4, 5), (6,7)));
```

它创建一个叫 Maze 的数组，这里

```
Maze[0,0,0] = 0
Maze[0,0,1] = 1
Maze[0,1,0] = 2
Maze[0,1,1] = 3
Maze[1,0,0] = 4
Maze[1,0,1] = 5
Maze[1,1,0] = 6
Maze[1,1,1] = 7
```

数组常量不能包含文件类型的值。

Record constants（记录常量）

要声明一个记录常量，在括号中使用 fieldName: value 的形式来指定每个字段的值，每个字段用分号隔开。每个字段的值必须是常量表达式。字段列出的顺序必须和声明的相同，若有 tag 字段，则必须指定它的值；若记录有一个 Variant 部分，只有 tag 字段选定的 Variant 才能被赋值。
举例如下：

```
type
  TPoint = record
    X, Y: Single;
  end;
  TVector = array[0..1] of TPoint;
  TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
  TDate = record
    D: 1..31;
    M: TMonth;
    Y: 1900..1999;
  end;
const
  Origin: TPoint = (X: 0.0; Y: 0.0);
  Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
  SomeDay: TDate = (D: 2; M: Dec; Y: 1960);
```

记录常量不能包含文件类型的值。

Procedural constants（过程常量）

要声明一个过程常量，指定一个和（常量）声明的类型兼容的函数或过程的名字。比如，

```
function Calc(X, Y: Integer): Integer;
begin
 …
end;
type TFunction = function(X, Y: Integer): Integer;
const MyFunction: TFunction = Calc;
```

给定这些声明，在调用函数时你能使用过程常量 MyFunction：

```
I := MyFunction(5, 7)
```

你也可以给一个过程常量赋 **nil** 值。

Pointer constants（指针常量）

当声明一个指针常量时，你必须给它初始化一个在编译时就能被解析的值，至少是一个相对地址。这有三种方式：使用@运算符、使用 **nil**、和字符串常量（若常量是 PChar 类型）。比如，若 I 是一个 Integer 类型的全局变量，你可以这样声明一个常量

```
const PI: ^Integer = @I;
```

因为全局变量是代码段（code segment）的一部分，编译器能解析它的值，函数和全局常量也是一样。

```
const PF: Pointer = @MyFunction;
```

因为常量字符串被当作全局常量进行分配，所以你能使用常量字符串来初始化一个 PChar 常量：

```
const WarningStr: PChar = 'Warning!';
```

局部变量（在堆栈中分配）和动态变量（在堆中分配）的地址不能赋给指针常量。

Procedures and functions（过程和函数）

Procedures and functions: Overview（概述）

过程和函数统称为例程（routine），它们是自包含的语句块，能在程序的不同地方调用。函数在在执行时能返回一个值，而过程不能。

因为函数调用能返回值，它们能被用在赋值和运算表达式中。比如，

```
I := SomeFunction(X);
```

调用 SomeFunction 并且把它的结果赋给 I。函数调用不能出现在赋值语句的左边。

过程调用能被用作一个完整的语句，并且，当启用扩展语法时（**{$X+}**），函数调用也可以。比如，

```
DoSomething;
```

调用 DoSomething 例程，若 DoSomething 是一个函数，它的返回值被忽略。

过程和函数能递归调用。

Declaring procedures and functions（声明过程和函数）

Declaring procedures and functions: Overview（概述）

声明过程或函数时，要指定它的名称，它使用的参数（数目和类型），如果是函数的话，还有返回值的类型，这一部分有时被称为原型、标题或头（prototype、heading、header）。然后你为它写一个代码块，当过程或函数被调用时，它们将会执行。这一部分有时称为例程体或例程块（body、block）。

标准过程 Exit 可出现在任何过程或函数中，它结束例程的执行，并立即把程序控制返回到例程调用的地方。

Procedure declarations（过程声明）

一个过程声明有如下格式：

```
procedure procedureName(parameterList); directives;
localDeclarations;
begin
  statements
end;
```

这里，*procedureName* 是任何有效标志符；*statements* 是一系列语句，当调用过程时它们被执行；(*parameterList*)、*directives*; 和 *localDeclarations* 是可选的。

- 要了解 *parameterList* 的信息，请参考 Parameters；
- 要了解 *directives* 的信息，请参考 Calling conventions、Forward and interface declarations、External declarations、 Overloading procedures and functions、和 Writing dynamically loadable libraries。如果包含多个指示字，用分号把它们隔开；
- *localDeclarations* 定义局部标志符，要了解它的信息，请参考 Local declarations。

这里是一个过程声明的例子：

```
procedure NumString(N: Integer; var S: string);
var
```

```
    V: Integer;
  begin
    V := Abs(N);
    S := '';
    repeat
      S := Chr(V mod 10 + Ord('0')) + S;
      V := V div 10;
    until V = 0;
    if N < 0 then S := '-' + S;
  end;
```

给定上面的声明，你能像这样调用 NumString 过程：

```
  NumString(17, MyString);
```

这个过程调用把值“17”赋给 MyString（它必须是字符串变量）。

在过程的语句块中，你可以使用在 *localDeclarations* 部分声明的变量和其它标志符；你也能使用参数列表中的参数名称(像上面例子中的 N 和 S)。参数列表定义了一组局部变量，所以，不要在 *localDeclarations* 部分重新声明它们；最后，你还可以使用过程声明所在范围中的任何标志符。

（以下内容摘自《**Delphi** 技术手册》

有些过程并不是普通的过程而是内置在编译器之中的，所以你无法获得它们的地址。一些过程（如 Exit）用起来就像是语言中的语句一样，但它们不是保留的关键字，你可以像使用其它过程一样使用它们。）

Function declarations（函数声明）

函数声明和过程声明类似，除了它要指定一个返回值的类似和返回值。函数声明有如下格式：

```
  function functionName(parameterList): returnType; directives;
  localDeclarations;
  begin
    statements
  end;
```

这里，functionName 是任何有效标志符，returnType 是任何类型，statements 是一系列语句，当调用函数时它们被执行；(*parameterList*)、*directives*; 和 *localDeclarations* 是可选的。

- 要了解 *parameterList* 的信息，请参考 Parameters；
- 要了解 *directives* 的信息，请参考 Calling conventions、Forward and interface declarations、External declarations、 Overloading procedures and functions、和 Writing dynamically loadable libraries。如果包含多个指示字，用分号把它们隔开；
- *localDeclarations* 定义局部标志符，要了解它的信息，请参考 Local declarations。

函数的语句块和过程遵循相同的规则：在语句块内部，你可以使用在 *localDeclarations* 部分声明的变量和其它标志符、参数列表中的参数名称，以及函数声明所在范围的所有标志符。除此之外，函数名本身也扮演一个特殊的变量，它和内置的变量 Result 一样，存储函数的返回值。

比如，

```
  function WF: Integer;
  begin
    WF := 17;
  end;
```

定义了一个叫做 WF 的常量函数，它没有任何参数，并且返回值总是 17。它和下面的声明是相同的：

```
  function WF: Integer;
```

```
begin
  Result := 17;
end;
```

下面是一个更复杂的函数声明：

```
function Max(A: array of Real; N: Integer): Real;
var
  X: Real;
  I: Integer;
begin
  X := A[0];
  for I := 1 to N - 1 do
    if X < A[I] then X := A[I];
  Max := X;
end;
```

在语句块内部，你可以重复给 Result 或函数名赋值，只要这个值的类型和函数声明的返回值类型相同即可。当函数的执行结束时，Result 或函数名最后被赋予的值当作函数的返回值。比如，

```
function Power(X: Real; Y: Integer): Real;
var
  I: Integer;
begin
  Result := 1.0;
  I := Y;
  while I > 0 do
  begin
    if Odd(I) then Result := Result * X;
    I := I div 2;
    X := Sqr(X);
  end;
end;
```

Result 和函数名总是表示同一个值，因此

```
function MyFunction: Integer;
begin
  MyFunction := 5;
  Result := Result * 2;
  MyFunction := Result + 1;
end;
```

返回值 11。但 Result 和函数名并不是能完全互换的，当函数名出现在赋值语句的左边时，编译器假设它用来跟踪（存储）返回值（就像 Result）；在任何其它情况下，编译器把它解释为对它的递归调用。而对 Result，它可以作为变量用在运算、类型转换、集合构造器、索引以及调用其它例程。

只要启用了扩展语法（**{$X+}**），Result 在每个函数中被隐含声明，不要试图重新声明它。

若还没有给 Result 或函数名赋值，程序就结束了，则函数的返回值没有被定义（undefined）。

（**以下内容摘自《Delphi 技术手册》**

并不是所有的函数都是真正的函数；有些是内置在编译器中的。这种差别通常并不重要，因为内置函数看上去和用起来都像是普通的函数，但是你无法获得内置函数的地址。）

Calling conventions（调用约定）

在声明过程或函数时，你可以使用下面的指示字之一来指明调用约定：**register**、**pascal**、**cdecl**、**stdcall** 以及 **safecall**。比如，

```
function MyFunction(X, Y: Real): Real; cdecl;
 …
```

调用约定决定了参数被传递给例程的顺序，它们也影响从堆栈中删除参数、传递参数时寄存器的使用，以及错误和异常处理。默认的调用约定是 **register**。

- **register** 和 **pascal** 调用从左到右传递参数，也就是说，最左边的参数最早被计算并传递，最右边的参数最后被计算和传递；**cdecl**、**stdcall** 和 **safecall** 调用从右到左传递参数；
- 除了 **cdecl** 调用，过程和函数在返回之前从堆栈中移除参数，而使用 **cdecl**，当调用返回时，调用者从堆栈中移除参数；
- **register** 调用能使用多达 3 个 CPU 寄存器传递参数，而其它调用则全部使用堆栈传递参数；
- **safecall** 调用实现了异常"防火墙"，在 Windows 下，它实现了进程间 COM 错误通知。

下面的表格对调用约定进行了总结：

指示字	参数顺序	**Clean-up**	使用寄存器传递参数?
register	Left-to-right	Routine	Yes
pascal	Left-to-right	Routine	No
cdecl	Right-to-left	Caller	No
stdcall	Right-to-left	Routine	No
safecall	Right-to-left	Routine	No

默认的 **register** 调用是最有效的，因为它通常避免了要创建堆栈结构（stack frame）（访问公布属性的方法必须使用 **register**）；当调用来自 C/C++编写的共享库中的函数时，**cdecl** 是有用的；通常，当调用外部代码时，推荐使用 **stdcall** 和 **safecall**。在 Windows 中，系统 API 使用 **stdcall** 和 **safecall**，其它操作系统通常使用 **cdecl**（注意，**stdcall** 比 **cdecl** 更有效）。

声明双重接口的方法必须使用 **safecall**；保留 **pascal** 调用是为了向后兼容性。要了解更多的调用约定的信息，请参考 Program control。

指示字 **near**、**far** 和 **export** 用在 16 位 Windows 编程中，它们对 32 位程序没有影响，保留它们是为了向后兼容性。

Forward and interface declarations（Forward 声明和接口部分的声明）

在声明过程或函数时，用 **forward** 指示字取代例程块（包括局部变量声明和语句），比如，

```
function Calculate(X, Y: Integer): Real; forward;
```

这就声明了一个叫做 Calculate 的函数，在 **forward** 声明后的某个地方，例程必须进行定义声明，包括例程块。Calculate 的定义声明看起来这样：

```
function Calculate;
 …    { declarations }
begin
 …    { statement block }
end;
```

通常，定义声明不必重新列出例程使用的参数或者返回值类型，但如果这样做的话，它必须和 **forward** 声明完全一致（除了默认参数能被忽略）。若 **forward** 声明是一个重载的过程或函数，则必须在定义声明中重新列出它的参数。

在 **forward** 声明和它的定义声明之间，除了声明外不能有其它内容。定义声明可以是 **external** 或 **assembler**，但不能是另外的 **forward** 声明。

forward 声明的目的是把过程或函数标志符的作用域提前，这允许在它被实际定义之前，其它过程和函数可以进行调用。除了能使你更灵活地组织代码外，**forward** 声明对相互递归调用（mutual recursion）有时是必须的。

在单元的接口（interface）部分，**forward** 指示字是不允许的，但是，接口部分的过程头或函数头，它们的行为就像 forward 声明，它们的定义声明必须出现在实现（implementation）部分。在单元的接口部分声明的例程，在本单元的其它任何位置都可以访问；对于使用本单元的其它单元或程序，这些例程也是可以访问的。

External declarations（External 声明）

在声明过程或函数时，用 **external** 指示字取代例程块，能允许你调用和程序分开编译的例程。外部例程可以来自目标文件或动态调入库（dynamically loadable library）。

当导入一个带有可变数目参数的 C++函数时，要使用 **varargs** 指示字。比如，

```
function printf(Format: PChar): Integer; cdecl; varargs;
```

varargs 指示字只能用于外部例程，并且只能使用 **cdecl** 调用约定。

链接目标文件

要调用目标文件中的例程，首先要使用**$L**（或**$LINK**）编译器指示字把目标文件链接到你的程序中。比如，

在 Windows 下： {$L BLOCK.OBJ}

在 Linux 下： {$L block.o}

把 BLOCK.OBJ（Windows）或 block.o （Linux）链接到程序或它所在的单元。然后，声明你想调用的函数和过程：

```
procedure MoveWord(var Source, Dest; Count: Integer); external;
procedure FillWord(var Dest; Data: Integer; Count: Integer); external;
```

现在，你就能调用来自 BLOCK.OBJ 或 block.o 的 MoveWord 和 FillWord 例程了。

像上面的声明，经常用来访问由汇编语言编写的外部例程，你也可以直接在 Object Pascal 源代码中放置汇编语言写的例程。关于这方面的信息，请参考 Inline assembler code。

从库导入函数

要从一个动态调入库（.so 或.DLL）导入例程，把如下格式的指示字

```
external stringConstant;
```

放在一个正常的过程头或函数头的尾部。这里，*stringConstant* 是用单引号括起来的库文件的名称。比如，在 Windwos 下

```
function SomeFunction(S: string): string; external 'strlib.dll';
```

从 strlib.dll 导入一个叫做 SomeFunction 的函数。

在 Linux 下，

```
function SomeFunction(S: string): string; external 'strlib.so';
```

从 strlib.so 导入一个叫做 SomeFunction 的函数。

在导入例程时，它的名称可以和库中的名称不同。如果你这样做，在 **external** 指示字中指定它的原始名称。

```
external stringConstant1 name stringConstant2;
```

这里，第一个 stringConstant 给出了库文件的名称，第二个 stringConstant 是例程的原始名称。

在 Windows 下：比如，下面的声明从 user32.dll（Windows API 的一部分）导入一个函数。

```
function MessageBox(HWnd: Integer; Text, Caption: PChar; Flags: Integer): Integer;
  stdcall; external 'user32.dll' name 'MessageBoxA';
```

函数的原始名称是 MessageBoxA，但导入后的名称是 MessageBox。

你可以使用一个数字代替名称，来指定你要导入的例程：

```
external stringConstant index integerConstant;
```

这里，integerConstant 是输出表（export table）中例程的索引。

在 Linux 下：比如，下面的声明从 libc.so.6 导入一个标准系统函数。

```
function OpenFile(const PathName: PChar; Flags: Integer): Integer; cdecl;
  external 'libc.so.6' name 'open';
```

函数的原始名称是 open，但导入后的名称是 OpenFile。

在你的导入声明中，要保证例程的名称没有拼写错误，并且大小写一致。但在以后调用这些例程时，它们是不区分大小写的。

关于库的更多信息，请参考 Libraries and packages。

Overloading procedures and functions（重载过程和函数）

你能使用相同的名称在一个作用域声明多个例程，这叫做重载。重载例程必须使用 overload 指示字，并且它们有不同的参数列表。比如，考虑下面的声明

```
function Divide(X, Y: Real): Real; overload;
begin
  Result := X/Y;
end;
function Divide(X, Y: Integer): Integer; overload;
begin
  Result := X div Y;
end;
```

这些声明创建了两个函数，它们都叫做 Divide，使用不同类型的参数。当调用 Divide 时，编译器通过查看实参的类型来决定哪个函数被调用。比如，Divide(6.0, 3.0)调用第一个 Divide 函数，因为它的参数是实数类型。

You can pass to an overloaded routine parameters that are not identical in type with those in any of the routine’s declarations, but that are assignment-compatible with the parameters in more than one declaration. This happens most frequently when a routine is overloaded with different integer types or different real types－for example,

你能给一个重载的例程传递这样的参数：它们的类型和所有例程声明中的参数类型都不同，但又和多个例程声明中的参数是赋值兼容的。对于使用了不同的整数（或实数）类型的重载例程来说，这种情况最经常发生，比如，

```
procedure Store(X: Longint); overload;
procedure Store(X: Shortint); overload;
```

在这种情况下，只要不产生模棱两可，编译器将调用如下的例程：传递给它的实参和它声明的参数相匹配，并且声明的参数范围最小。（记住，实数相关的常量值总是 **Extended** 类型。）

重载例程必须能以参数的数目或类型区分开来，因此，下面的声明将产生编译错误：

```
function Cap(S: string): string; overload;
 ...
procedure Cap(var Str: string); overload;
```

...
但声明

```
function Func(X: Real; Y: Integer): Real; overload;
 ...
function Func(X: Integer; Y: Real): Real; overload;
 ...
```

是合法的。
当重载例程被声明为 **forward**、或在单元的接口部分声明时，在它的定义声明部分必须重新列出它的参数。
编译器能区分两个包含 AnsiString/Pchar 以及 WideString/WideChar 的重载例程。在这种情况下，字符串常量或文字被解释为 native 字符串或字符类型，也就是 AnsiString/Pchar。

```
procedure test(const S: String);  overload;
procedure test(const W: WideString); overload;
var
    a: string;
    b: widestring;
begin
   a := 'a';
   b := 'b';
   test(a);                      // 调用 String 版本
   test(b);                      // 调用 widestring 版本
   test('abc');                  // 调用  String 版本
   test(WideString('abc'));      // 调用 widestring 版本
end;
```

在声明重载例程时能使用 Variant 类型的参数。Variant 类型被认为比简单类型更通用，精确的类型匹配比 Variant 更有优先权。If a Variant is passed into such an overload situation, and an overload that takes a Variant exists in that parameter position, it is considered to be an exact match for the Variant type.
这对浮点类型有些影响。浮点类型通过大小进行匹配，如果在调用时，参数类型和传递的浮点类型不完全匹配、但有一个 Variant 类型的参数，则采用 Variant 类型，而不是其它较小的浮点类型（the Variant is taken over any smaller float type）。

比如

```
procedure foo(i: integer); overload;
procedure foo(d: double); overload;
procedure foo(v: Variant); overload;
var
  v: Variant;
begin
  foo(1);         // integer 版本
  foo(v);         // Variant 版本
  foo(1.2);       // Variant 版本(float literals -> extended 精度)
end;
```

这个例子调用 foo 的 Variant 版本，而不是 double 版本，因为常数 1.2 隐含是 extended 类型，它和 double 类型不完全匹配。Extended 和 Variant 类型也不是完全匹配，但 Variant 类型更通用（尽管 double 类型比 extended 类型更小）。

```
foo(Double(1.2));
```

这个类型转换不起作用，你应该使用指定类型的常数。

```
const  d: double = 1.2;
  begin
     foo(d);
  end;
```

上面的代码工作正常，将调用 double 版本。

```
const  s: single = 1.2;
begin
   foo(s);
end;
```

上面的代码也是调用 double 版本，相对于 Variant，single 和 double 更匹配，

当声明一组重载例程时，避免浮点类型解释为 Variant 类型的最好办法，是在声明 Variant 版本的同时，为每一种浮点类型（Single、Double、Extended）声明一个版本。

若在重载例程中使用默认参数，要当心不明确的参数调用。关于更多这方面的信息，请参考 Default parameters and overloaded routines。

通过在调用例程时限定它的名称，能减少重载例程潜在的影响。比如，Unit1.MyProcedure(X, Y)只能调用在 Unit1 单元声明的例程，若 Unit1 中没有和指定的名称以及参数相匹配的例程，错误将发生。

关于在类的继承层次中发布重载方法，请参考 Overloading methods；关于在共享库中输出重载例程，请参考 The exports clause。

Local declarations（局部声明）

函数体或过程体经常以声明局部变量开始，它们用在例程的语句块中。这里也可以包括常量、类型以及例程声明。局部标志符的作用域被限制在声明它的例程中。

嵌套例程

有时，函数和过程在它的局部声明块中包含其它函数和过程。比如，下面声明了一个叫做 DoSomething 的过程，它包含一个嵌套过程。

```
procedure DoSomething(S: string);
var
  X, Y: Integer;
  procedure NestedProc(S: string);
  begin
    …
  end;
begin
   …
  NestedProc(S);
   …
end;
```

嵌套例程的作用域限制在声明它的过程或函数中，在我们的例子中，NestedProc 只能在 DoSomething 内部调用。

关于嵌套例程的实际应用，请参考 SysUtils 单元的 DateTimeToString 过程、ScanDate 函数以及其它例程。

Parameters（参数）

Parameters: Overview（概述）

大多数过程头和函数头包含参数列表，比如，在函数头

```
function Power(X: Real; Y: Integer): Real;
```

中，参数列表是(X: Real; Y: Integer)。

参数列表由一对括号包围，包含由分号隔开的一系列参数声明。每个声明中列出的参数名由逗号分开，大多数情况下后面跟一个冒号和一个类型标志符，在一些情况下跟一个等号和一个默认值。参数名必须是有效标志符。任何一个声明能以下面的一个关键字作为前缀：**var**、**const** 和 **out**（参考 Parameter semantics）。比如：

```
(X, Y: Real)
(var S: string; X: Integer)
(HWnd: Integer; Text, Caption: PChar; Flags: Integer)
(const P; I: Integer)
```

参数列表指定了调用例程时必须传递的参数的数目、顺序和类型。若一个例程没有任何参数，声明时忽略括号和标志符列表。

```
procedure UpdateRecords;
begin
  …
end;
```

在过程体或函数体中，参数名（上面第一个例子中的 X 和 Y）能被用作局部变量，在局部声明中不能重新声明参数名。

Parameter semantics（参数语义）

Parameter semantics: Overview（概述）

参数以下面几种方式进行分类：

- 每个参数分为 value（数值参数）、variable（变量参数）、constant（常量参数）或 out（out 参数），默认是数值参数。关键字 var、const 以及 out 分别表示变量参数、常量参数和 out 参数。
- 数值参数总是有类型的，而常量参数、变量参数和 out 参数既可以是有类型的，也可以是无类型的。
- 数组参数有特殊规则。

文件类型以及包含文件的结构类型（的实例）只能作为变量参数传递。

Value and variable parameters（数值参数和变量参数）

大多数参数是数值参数（默认）或变量参数（var）。数值参数通过数值传递，而变量参数通过引用传递。要了解这句话的意义，考虑下面的函数。

```
function DoubleByValue(X: Integer): Integer;          // X 是数值参数
begin
  X := X * 2;
```

```
    Result := X;
  end;
  function DoubleByRef(var X: Integer): Integer;        // X 是变量参数
  begin
    X := X * 2;
    Result := X;
  end;
```

这两个函数返回同样的结果，但只有第二个（DoubleByRef）能改变传给它的变量的值。假设我们这样调用函数：

```
  var
    I, J, V, W: Integer;
  begin
    I := 4;
    V := 4;
    J := DoubleByValue(I);       // J = 8, I = 4
    W := DoubleByRef(V);         // W = 8, V = 8
  end;
```

这些代码执行后，传给 DoubleByValue 的变量 I，它的值和我们初始赋给它的值是相同的。但传给 DoubleByRef 的变量 V，它有不同的值。

数值参数就像局部变量，它们的初始值是传给过程或函数的值。若把一个变量当作数值参数传递，过程或函数创建它的一个拷贝，改变这个拷贝对原始变量没有影响，并且，当程序返回调用者时，这个拷贝将被丢弃。

而另一方面，变量参数就像一个指针而不是一个拷贝，当程序返回调用者时，在函数或过程体中对它的改变将被保留，（仅仅）参数名本身超出了作用域。

即使同一个变量被传给两个或多个参数，也不会创建它的拷贝，这通过下面的例子说明。

```
  procedure AddOne(var X, Y: Integer);
  begin
    X := X + 1;
    Y := Y + 1;
  end;
  var I: Integer;
  begin
    I := 1;
    AddOne(I, I);
  end;
```

这些代码执行后，I 的值是 3。

如果例程声明了一个 **var** 参数，你必须给它传递一个能被赋值的表达式，也就是一个变量、类型化常量（typed constant，在{$J+}状态下）、dereferenced 指针、字段或者索引变量（indexed variable）。在前面的例子中，DoubleByRef(7)会产生错误，而 DoubleByValue(7)是合法的。

索引以及指针引用被当作 **var** 参数时，在例程执行之前它们会被计算（一次）。

Constant parameters（常量参数）

一个常量参数（**const**）就像一个局部常量或者一个只读变量，常量参数和数值参数类似，但在过程或函数中你不能给常量参数赋值，也不能把它当作变量参数传给另一个例程。（但是，当把一个对象引用当作

常量参数时，你仍然可以修改对象的属性。）
使用 **const** 允许编译器对结构类型和字符串类型的参数做代码优化，也防止不小心把一个参数通过引用传给另一个例程。
这里的例子是 SysUtils 单元中的 CompareStr 函数：

```
function CompareStr(const S1, S2: string): Integer;
```

因为 S1 和 S2 在 CompareStr 中没有被修改，它们可以被声明为常量参数。

Out parameters（Out 参数）

Out 参数和变量参数类似，通过引用传递。但是，当使用 out 参数时，传给例程的引用参数的初始值被忽略。out 参数只是为了输出，也就是说，它告诉函数或过程在哪里存储输出，但不提供任何输入。
比如，考虑过程头声明

```
procedure GetInfo(out Info: SomeRecordType);
```

当你调用 GetInfo 时，你必须给它传递一个 SomeRecordType 类型的变量：

```
var MyRecord: SomeRecordType;
 ...
GetInfo(MyRecord);
```

但你不使用 MyRecord 给 GetInfo 传递任何值，MyRecord 只是一个容器，你希望 GetInfo 用它存储得到的信息。The call to GetInfo immediately frees（释放还是清零？） the memory used by MyRecord, before program control passes to the procedure.
out 参数经常用在分布式对象模型中，比如 COM 和 CORBA。而且，当向函数或过程传递未初始化的变量时，你应当使用 out 参数。

Untyed parameters（无类型参数）

当声明 **var**、**const** 和 **out** 参数时，你可以省略类型说明（数值参数必须指定类型）。比如，

```
procedure TakeAnything(const C);
```

声明一个叫做 TakeAnything 的过程，它可以接受任何类型的参数。当你调用这样一个例程时，你不能向它传递 numeral or untyped numeric constant。
在过程体或函数体中，无类型参数和每个类型都不兼容。要对一个无类型参数进行操作，你必须对它进行转换。通常，编译器不会对无类型参数检验它的有效性。
下面的例子在 Equal 函数中使用无类型参数，这个函数比较两个参数中指定数目的字节。

```
function Equal(var Source, Dest; Size: Integer): Boolean;
type
  TBytes = array[0..MaxInt - 1] of Byte;
var
  N: Integer;
begin
  N := 0;
  while (N < Size) and (TBytes(Dest)[N] = TBytes(Source)[N]) do
    Inc(N);
  Equal := N = Size;
end;
```

给定下面的声明

```
type
```

```
    TVector = array[1..10] of Integer;
    TPoint = record
      X, Y: Integer;
    end;
  var
    Vec1, Vec2: TVector;
    N: Integer;
    P: TPoint;
```

你可以如下调用 Equal：

```
Equal(Vec1, Vec2, SizeOf(TVector))            // 比较 Vec1 和 Vec2
Equal(Vec1, Vec2, SizeOf(Integer) * N)        // 比较 Vec1 和 Vec2 的前 N 个元素
Equal(Vec1[1], Vec1[6], SizeOf(Integer) * 5)  // 比较 Vec1 的前 5 个元素和后 5 个元素
Equal(Vec1[1], P, 4)                          // 比较 Vec1[1]和 P.X、Vec1[2]和 P.Y
```

String parameters（字符串参数）

About string parameters（关于字符串参数）

当例程使用短字符串（short-string）参数时，在声明时不能指定它们的长度。也就是说，声明

```
  procedure Check(S: string[20]);    // 语法错误
```

产生编译错误。但是

```
  type TString20 = string[20];
  procedure Check(S: TString20);
```

是有效的。特殊标志符 *OpenString* 能用于声明可变长度的短字符串参数：

```
  procedure Check(S: OpenString);
```

当编译器指示字**{$H-}**和**{$P+}**都起作用时，在声明参数时关键字 **string** 等同于 *OpenString*。

短字符串、OpenString、$H 和$P 是为了向后兼容性。在新代码中，使用长字符串来避免这种情况。

Array parameters（数组参数）

Array parameters: Overview（概述）

当例程使用数组参数时，你不能在声明参数时包含索引说明符。也就是说，声明

```
  procedure Sort(A: array[1..10] of Integer);    // 语法错误
```

产生编译错误，但

```
  type TDigits = array[1..10] of Integer;
  procedure Sort(A: TDigits);
```

是有效的。但在大多数情况下，开放数组参数是更好的办法。

Open array parameters（开放数组参数）

开放数组参数允许传递不同大小的数组到同一个过程或函数。要定义一个使用开放数组参数的例程，在

声明参数时使用 array of *type*（而不是 array[X..Y]）。

```
function Find(A: array of Char): Integer;
```

声明了一个叫做 Find 的函数，它接收任意长度的字符数组并返回一个整数。

注意：开放数组参数的语法和声明动态数组相似，但它们的意义不同。上面的例子创建了一个函数，它可以接收由字符构成的任何数组，包括（但不限于）动态数组。对于必须是动态数组的参数，你需要指定一个类型标志符：

```
type TDynamicCharArray = array of Char;
function Find(A: TDynamicCharArray): Integer;
```

关于动态数组的信息，请参考 Dynamic arrays。

在例程体（body）中，开放数组参数遵循下列规则：

- 元素的下标总是从 0 开始，第一个是 0，第二个是 1，依此类推。标准函数 Low 和 High 返回 0 和 Length-1。SizeOf 函数返回传给例程的实际数组的大小；
- 它们只能通过元素进行访问，不允许给整个开放数组赋值；
- 它们只能被当作开放数组参数或无类型 **var** 参数传给其它过程和函数，它们不能传给 SetLength 函数；
- 你可以传递一个变量而不是数组，变量的类型就是开放数组的基础类型，它被当作一个长度为 1 的数组。

当把一个数组当作开放数组数值参数传递时，编译器在例程的堆栈结构（stack frame）中创建一个本地拷贝，传递大数组时要当心堆栈溢出。

下面的例子使用开放数组参数定义了一个 Clear 过程，它把数组中的每个实数元素赋 0；还定义了一个 Sum 函数，它计算实数数组的元素之和。

```
procedure Clear(var A: array of Real);
var
  I: Integer;
begin
  for I := 0 to High(A) do A[I] := 0;
end;
function Sum(const A: array of Real): Real;
var
  I: Integer;
  S: Real;
begin
  S := 0;
  for I := 0 to High(A) do S := S + A[I];
  Sum := S;
end;
```

当调用使用开放数组参数的例程时，你可以向它传递开放数组构造器。

Variant open array parameters（Variant 开放数组参数）

Variant 开放数组参数允许你向一个过程或函数传递由不同类型的元素构成的数组。要定义这样一个例程，指定 **array of const** 作为参数的类型，这样

```
procedure DoSomething(A: array of const);
```

声明了一个叫做 DoSomething 的过程，它能接收不同类型的数组。

array of const 结构等同于 **array of** TVarRec。TVarRec 在 System 单元定义，表示一个拥有变体部分的记

录，它能存储整数、布尔、字符、实数、字符串、指针、类、类引用、接口和变体类型的值。TVarRec 记录的 VType 字段指示数组中每个元素的类型。一些类型以指针而不是以数值形式进行传递，特别是，长字符串以指针类型传递，必须被转换为 string。
下面的函数使用 Variant 开放数组参数，它把传给它的每个元素用字符串表示，并把它们连接成一个单一的字符串。函数中使用的字符串处理例程在 SysUtils 单元定义。

```
function MakeStr(const Args: array of const): string;
const
  BoolChars: array[Boolean] of Char = ('F', 'T');
var
  I: Integer;
begin
  Result := '';
  for I := 0 to High(Args) do
    with Args[I] do
      case VType of
        vtInteger:    Result := Result + IntToStr(VInteger);
        vtBoolean:    Result := Result + BoolChars[VBoolean];
        vtChar:       Result := Result + VChar;
        vtExtended:   Result := Result + FloatToStr(VExtended^);
        vtString:     Result := Result + VString^;
        vtPChar:      Result := Result + VPChar;
        vtObject:     Result := Result + VObject.ClassName;
        vtClass:      Result := Result + VClass.ClassName;
        vtAnsiString: Result := Result + string(VAnsiString);
        vtCurrency:   Result := Result + CurrToStr(VCurrency^);
        vtVariant:    Result := Result + string(VVariant^);
        vtInt64:      Result := Result + IntToStr(VInt64^);
    end;
end;
```

我们可以传递一个开放数组构造器来调用这个函数（参考 Open array constructors）。比如，

```
MakeStr(['test', 100, ' ', True, 3.14159, TForm])
```

返回字符串“test100 T3.14159TForm”。

Default parameters（默认参数）

Default parameters（默认参数）

在过程头或函数头中，你可以指定默认的参数值。默认值只允许出现在指定类型的常量参数和数值参数中。要提供一个默认值，在参数声明中以等号和一个常量表达式（和参数的类型赋值兼容）作为结束。比如，给定下面的声明

```
procedure FillArray(A: array of Integer; Value: Integer = 0);
```

下面的过程调用是相等的。

```
FillArray(MyArray);
FillArray(MyArray, 0);
```

一个多参数声明不能指定一个默认值，这样，虽然

```
function MyFunction(X: Real = 3.5; Y: Real = 3.5): Real;
```

是合法的，但

```
function MyFunction(X, Y: Real = 3.5): Real;    // 语法错误
```

是非法的。

有默认值的参数必须出现在参数列表的最后，也就是说，在声明了一个有默认值的参数后，它后面的所有参数也必须有默认值。所以，下面的声明是非法的。

```
procedure MyProcedure(I: Integer = 1; S: string);    // 语法错误
```

在过程类型中指定的默认值会覆盖实际例程中指定的默认值。所以，给出下面的声明

```
type TResizer = function(X: Real; Y: Real = 1.0): Real;
function Resizer(X: Real; Y: Real = 2.0): Real;
var
  F: TResizer;
  N: Real;
```

语句

```
F := Resizer;
F(N);
```

导致(N, 1.0)传给 Resizer。

默认参数局限于能被常量表达式所表示的值，所以，动态数组、过程、类、类引用或者接口类型的参数除了 **nil** 外不能给它们指定默认值，而记录、变体、文件、静态数组和对象类型则根本不能指定默认值。

关于调用有默认值的例程，请参考 Calling procedures and functions。

Default parameters and overloaded routines（默认参数和重载例程）

若在重载例程中使用默认参数，要避免引起歧义。比如，考虑下面的代码

```
procedure Confused(I: Integer); overload;
 ...
procedure Confused(I: Integer; J: Integer = 0); overload;
 ...
Confused(X);     //  要调用哪一个呢？
```

实际上，哪个过程都不会调用，代码产生编译错误。

Default parameters in forward and interface declarations（forward 声明中的默认参数）

若例程是 **forward** 声明或出现在单元的接口部分，则默认参数值（若有的话）必须在 forward 声明或在接口部分的声明中指定。这种情况下，定义（实现）声明中的默认值可以被忽略，但如果定义声明包含默认值的话，它们必须和 **forward** 声明或接口部分的声明一致。

Calling procedures and functions（调用过程和函数）

Calling procedures and functions（调用过程和函数）

当调用一个过程或函数时，程序控制从函数的调用点转到例程体中。调用例程时，你可以使用例程的名称（带或不带限定符），也可以使用指向例程的过程变量。不管哪种情况，若例程声明使用参数，则调用时必须传递参数，并且它们的顺序和类型必须一致。传递给例程的参数叫做实参，而声明例程时的参数称为形参。

当调用例程时，记住

- 用来传递指定类型的常量参数和数值参数的表达式必须和相应的形参是赋值兼容的；
- 用来传递 **var** 和 **out** 参数的表达式必须和相应的形参类型相同，除非形参没有指定类型（无类型）；
- 只有能被赋值的表达式可用作 **var** 和 **out** 参数；
- 如果例程的形参是无类型的，numerals and true constants with numeric values 不能用作实参。

当调用使用默认参数的例程时，第一个默认参数后面的实参也必须使用默认值，像 SomeFunction(,,X)形式的调用是非法的。

当一个例程全部使用默认参数、并且都使用默认值调用时，可以省略它的括号。比如，给定过程声明

```
procedure DoSomething(X: Real = 1.0; I: Integer = 0; S: string = '');
```

下面的调用是等同的。

```
DoSomething();
DoSomething;
```

Open array constructors（开放数组构造器）

开放数组构造器允许你在函数和过程调用中直接构造数组，它们只能被当作开放数组参数或 Variant 开放数组参数进行传递。

开放数组构造器和集合构造器类似，是由逗号隔开的表达式序列，并且被一对中括号包围。

比如，给定声明

```
var I, J: Integer;
procedure Add(A: array of Integer);
```

你可以使用下面的语句调用 Add 过程

```
Add([5, 7, I, I + J]);
```

这等同于

```
var Temp: array[0..3] of Integer;
 ...
Temp[0] := 5;
Temp[1] := 7;
Temp[2] := I;
Temp[3] := I + J;
Add(Temp);
```

开放数组构造器只能当作数值参数或常量参数传递。构造器中的表达式必须和数组参数的基础类型是赋值兼容的。对于 **Variant** 开放数组参数，表达式可以是不同的类型。

Classes and objects（类和对象）

Classes and objects: Overview（概述）

类（或者类类型）定义了一个结构，它包括字段（也称为域）、方法和属性；类的实例叫做对象；类的字段、方法和属性被称为它的部件（components）或成员。

- 字段在本质上是一个对象的变量。和记录的字段类似，类的字段表示一个类实例的数据项；
- 方法是一个过程或者函数，它和类相关联。绝大多数方法作用在对象（也就是类的实例）上，其它一些方法（称为类方法）作用在类上面。
- 属性被看作是访问对象的数据的接口，对象的数据通常用字段来存储。属性有存取设定，它决定数据如何被读取或修改。从程序的其它地方（在对象本身以外）来看，属性在很大程度上就像一个字段（但本质上它相当于方法，比如在类的实例中并不为它分配内存）。

对象被动态分配一个内存块，内存结构由类类型决定。每个对象拥有类中所定义的每个字段的唯一拷贝，但一个类的所有实例共享相同的方法。对象分别通过称为**构造函数**和**析构函数**的方法创建和销毁。

一个类变量实际是一个指针，它引用一个对象（称它为对象引用），所以，多个变量可以指向同一个对象。像其它指针一样，一个类变量的值可以是 **nil**。虽然类变量是一个指针，但我们直接用它来表示一个对象，例如，SomeObject.Size := 100 是把对象的 Size 属性设为 100，你不能用下面的命令给它赋值：SomeObject^.Size := 100。

Class types（类类型）

About class types（关于类类型）

类类型必须在实例化之前进行声明并给定一个名称（不能在变量声明中定义一个类类型），你只能在程序（program）或单元（unit）的最外层声明类，而不能在过程或函数中声明。

一个类声明有如下格式

```
type className = class (ancestorClass)
  memberList
 end;
```

这里，*className* 是任何有效标志符，（*ancestorClass*）是可选的，*memberList* 声明类的各成员，也就是它的字段、方法和属性。若你省略了（*ancestorClass*），则新定义的类直接继承自内置的类 TObject。如果包含（*ancestorClass*）并且 *memberList* 是空的，你可以省略 end。一个类声明也可以包括它实现的接口列表，请参考 Implementing interfaces。

在类声明中，方法看起来就像函数（或过程）头，而没有函数（或过程）体。方法的定义出现在程序的其它地方。

比如，这里是 Classes 单元中 TMemoryStream 类的声明

```
type
TMemoryStream = class(TCustomMemoryStream)
  private
    FCapacity: Longint;
    procedure SetCapacity(NewCapacity: Longint);
  protected
```

```
    function Realloc(var NewCapacity: Longint): Pointer; virtual;
    property Capacity: Longint read FCapacity write SetCapacity;
  public
    destructor Destroy; override;
    procedure Clear;
    procedure LoadFromStream(Stream: TStream);
    procedure LoadFromFile(const FileName: string);
    procedure SetSize(NewSize: Longint); override;
    function Write(const Buffer; Count: Longint): Longint; override;
  end;
```

TMemoryStream 是 TStream（在 Classes 单元中）的后代，继承了它的大部分成员，但又定义（或重新定义）了几个方法和属性，包括它的析构（destructor）函数 Destroy。它的构造函数 Create 从 TObject 继承，没有任何改变，所以没有重新声明。每个成员被声明为 private、protected 或者 public（这个类没有 published 成员），关于这些术语的解释，请参考 Visibility of class members。

给定上面的声明，你可以像下面一样创建 TMemoryStream 的一个实例：

```
var stream: TMemoryStream;
stream := TMemoryStream.Create;
```

Inheritance and scope（继承和作用域）

Inheritance and scope（继承和作用域）

当你声明一个类时，可以指定它的父类，比如，

```
type TSomeControl = class(TControl);
```

定义了一个叫做 TSomeControl 的类，它继承自 TControl。一个类自动从它的父类继承所有的成员，且可以声明新成员，也可以重新定义继承下来的成员，但不能删除祖先类定义的成员。所以，TSomeControl 包含了在 TControl 以及它的每个祖先中定义的所有成员。

类成员标志符的作用域开始于它声明的地方，直到类声明的结束，并且扩展到它的所有子类声明的地方，以及类和子类的所有方法的定义区（也就是方法的定义部分）。

TObject and TClass（TObject 和 TClass）

类 TObject 在 System 单元声明，是所有其它类的最终祖先。TObject 只定义了少数方法，包括一个基本的构造函数和析构函数。除了 TObject，System 单元还声明了一个类引用类型 TClass。

```
TClass = class of TObject;
```

如果在类声明中没有指定父类，则它直接继承于 TObject，所以

```
type TMyClass = class
…
end;
```

等同于

```
type TMyClass = class(TObject)
 …
end;
```

后者可读性较好，推荐使用。

Compatibility of class types（类类型兼容性）

类和它的祖先类是赋值兼容的，所以，某个类类型的变量能引用它的任何子类类型的实例。比如，在下面的声明中

```
type
  TFigure = class(TObject);
  TRectangle = class(TFigure);
  TSquare = class(TRectangle);
var
  Fig: TFigure;
```

变量 Fig 能被赋予 TFigure、TRectangle 和 TSquare 类型的值。

Object types（Object 类型）

除了类类型，你可以使用如下语法声明一个 object 类型

```
type objectTypeName = object (ancestorObjectType)
  memberList
end;
```

这里，*objectTypeName* 是任何有效标志符，（*ancestorObjectType*）是可选的，*memberList* 声明字段、方法和属性。若（*ancestorObjectType*）被省略了，则新类型没有祖先。Object 类型不能有 published 成员。

因为 object 类型不是从 TObject 继承，它们没有内置的构造函数和析构函数，也没有其它方法。你能使用 New 过程创建 Object 类型的实例，并使用 Dispose 过程销毁它们，你也可以像使用记录一样，采用简单方式声明 object 类型的变量。

Object 类型只是为了向后兼容性，不推荐使用它们。

Visibility of class members（类成员的可见性）

Visibility of class members（类成员的可见性）

类的每个成员都有一个称为可见性的属性，我们用下面的关键字之一来表示它：**private**、**protected**、**public**、**published** 和 **automated**。比如

```
published property Color: TColor read GetColor write SetColor;
```

声明一个叫做 Color 的 published 属性。可见性决定了一个成员在哪些地方以及如何能被访问，**private** 表示最小程度的访问能力，**protected** 表示中等程度的访问能力，**public**、**published** 和 **automated** 表示最大程度的访问能力。

若声明一个成员时没有指定其可见性，则它和前面的成员拥有相同的可见性；若在类声明的开始没有指定可见性，当在**{$M+}**状态下编译类时（或者继承自一个在**{$M+}**状态下编译的类），它的默认可见性是 published，否则，它的可见性是 public。

为可读性考虑，最好在声明类时用可见性来组织成员：把所有的 private 成员组织在一起，接下来是所有的 protected 成员，依此类推。用这种方法，每个可见性关键字最多出现一次，并且标明了每个新段的开始。所以，一个典型的类声明应该像下面的形式：

```
type
  TMyClass = class(TControl)
```

```
private
 ... { private declarations here}
protected
 ... { protected declarations here }
public
 ... { public declarations here }
published
 ... { published declarations here }
end;
```

通过重新声明,你可以在派生类中增大一个成员的可见性,但你不能降低它的可见性。比如,一个 protected 属性在派生类中能被改变为 public，但不能改为 private。还有，published 成员在子类中不能改为 public。要了解更多信息，请参考 Property overrides and redeclarations。

Private, protected, and public members（私有、受保护和公有成员）

Private 成员在声明它的单元或程序之外是不可用的，换句话说，一个 private 方法不能从另一个模块（module）进行调用，也不能从另一个模块读取或写入一个私有的字段或属性。通过把相关类的声明放在一个模块中，可以使它们拥有访问其它类的私有成员的能力，同时又不会增大这些成员的访问范围。

Protected 成员在声明它的类的模块中是随处可用的，并且在它的派生类中也是可用的，而不管派生类出现在哪个模块。换句话说，在派生类的所有方法定义中，你可以调用 protected 方法，也能读取或写入 protected 字段或属性。只有在派生类的实现中才应用的成员通常使用 protected 属性。

对于 public 成员，只要能使用类的地方都是可用的。

Published members（公布的成员）

Published 成员和 **public** 成员具有相同的可见性，不同之处是 **published** 成员会产生 RTTI 信息。RTTI 使应用程序能动态查询一个对象的字段和属性，也能定位它的方法。RTTI 用于在存储文件和从文件导入时访问属性的值，也用于在 Object Inspector 中显示属性，并且能为一些特定属性（叫做事件）关联特定的方法（叫做事件处理程序）。

公布属性的数据类型受到限制，有序类型、字符串、类、接口和方法指针能被公布；当集合类型的基础类型是有序类型，并且上界和下界介于 0 到 31 之间时（换句话说，集合必须符合 byte、word 或 double word），集合类型也是可以公布的；除了 Real48，任何实数类型都是可以公布的；数组类型的属性（区别于数组属性，array properties）不能是公布的。

一些属性虽然是可以公布的，但不能完全支持流系统，它们包括：记录类型的属性、所有可公布类型的数组属性以及包含匿名值的枚举类型的属性。如果 published 属性属于前面所述的类型，Object Inspector 不能正确显示它们，并且使用流向磁盘操作时也不能保存它们的值。

所有方法都是可以公布的，但一个类不能使用相同的名字公布两个或以上数目的被重载的方法。只有当字段属于类或接口类型时，它才是可以公布的。

A class cannot have published members unless it is compiled in the {$M+} state or descends from a class compiled in the {$M+} state. Most classes with published members derive from TPersistent, which is compiled in the {$M+} state, so it is seldom necessary to use the $M directive.

除非一个类是在**{$M+}**状态下被编译，或者派生于一个在**{$M+}**状态下被编译的类，否则它不能有公布的成员。大多数具有公布成员的类继承自 TPersistent，而它是在**{$M+}**状态下被编译的，所以通常很少使用**$M** 编译器指示字。

Automated members（自动化成员）

Automated 成员和 public 成员具有相同的可见性，不同之处是 automated 成员会产生自动化类型信息（Automation type information，自动化服务器需要）。Automated 成员只出现在 Windows 类中，不推荐在 Linux 程序中使用。保留关键字 automated 是为了向后兼容性，ComObj 单元的 TAutoObject 类不使用自动化成员。

对声明为 automated 类型的方法和属性有以下限制：

- 所有属性、数组属性的参数、方法的参数以及函数的结果，它们的类型必须是自动化类型，包括 Byte、Currency、Real、Double、Longint、Integer、Single、Smallint、AnsiString、WideString、TDateTime、Variant、OleVariant、WordBool 和所有接口类型。
- 方法声明必须使用默认的 **register** 调用约定，它们可以是虚方法，但不能是动态方法。
- 属性声明可以包含访问限定符（读和写），但不能包含其它限定符（index、stored、default 和 nodefault）。访问限定符指定的方法必须使用默认的 **register** 调用约定，并且限定符不能使用字段。
- 属性声明必须指定一个类型，并且属性不支持覆盖（override）。

Automated 方法或属性声明中可以包含 **dispid** 指示字，但指定一个已经使用的 ID 会导致错误。

在 Windows 中，这个指示字的后面必须跟一个整数常数，它为成员指定一个 Automation dispatch ID。否则，编译器自动为它指定一个 ID，这个 ID 等于类（包括它的祖先类）的方法或属性使用的最大 ID 加上 1。关于自动化的更多信息，请参考 Automation objects。

Forward declarations and mutually dependent classes（Forward 声明和相互依赖的类）

若声明一个类时以 class 和分号结束，也就是有下面的格式，

```
type className = class;
```

在 class 后面没有列出父类，也没有成员列表，这是一个 forward 声明。Forward 声明的类必须在同一个声明区域进行定义声明，换句话说，在 forward 声明和它的定义声明之间除了类型声明外，不能有任何其它内容。

Forward 声明允许创建相互依赖的类，比如

```
type
  TFigure = class;          // forward 声明
  TDrawing = class
    Figure: TFigure;
     …
  end;
  TFigure = class           // 定义声明
    Drawing: TDrawing;
     …
  end;
```

不要把 forward 声明和继承自 TObject、不包含任何类成员的完整类声明混淆：

```
type
  TFirstClass = class;                // 这是 forward 声明
  TSecondClass = class                // 这是一个完整的类声明
    end;                              //
  TThirdClass = class(TObject);       // 这是一个完整的类声明
```

Fields（字段）

字段就像属于对象的一个变量，它可以是任何类型，包括类类型（也就是说，字段可以存储对象的引用）。字段通常具有 private 属性。
给类定义字段非常简单，就像声明变量一样。字段声明必须出现在属性声明和方法声明之前，比如，下面的声明创建了一个叫做 TNumber 的类，除了继承自 TObject 的方法之外，它有一个唯一的整数类型的成员 Int。

```
type TNumber = class
  Int: Integer;
end;
```

字段是静态绑定的，也就是说，它们的引用在编译时是固定的。要理解上面的意思，请考虑下面的代码：

```
type
  TAncestor = class
    Value: Integer;
  end;
  TDescendant = class(TAncestor)
    Value: string;                          // 隐藏了继承的 Value 字段
  end;
var
  MyObject: TAncestor;
begin
  MyObject := TDescendant.Create;
  MyObject.Value := 'Hello!';               // 错误
  TDescendant(MyObject).Value := 'Hello!';  // 工作正常
end;
```

虽然 MyObject 存储了 TDescendant 的一个实例，但它是以 TAncestor 声明的，所以，编译器把 MyObject.Value 解释为 TAncestor 声明的整数字段。不过要知道，在 TDescendant 对象中，这两个字段都是存在的，继承下来的字段被新字段隐藏了，但可以通过类型转换对它进行操作。

Methods（方法）

Methods: Overview（概述）

方法是一个和类相关联的过程或函数，调用一个方法需指定它作用的对象（若是类方法，则指定类），比如，

```
SomeObject.Free
```

调用 SomeObject 的 Free 方法。

Method declarations and implementations（方法声明和实现）

Method declarations and implementations（方法声明和实现）

在类声明中，方法看起来像过程头和函数头，工作起来像 forward 声明。在类声明后的某个地方（必须属于同一模块），每个方法必须有一个定义声明来实现它。比如，假设 TMyClass 类声明包含一个叫做 DoSomething 的方法：

```
type
  TMyClass = class(TObject)
     ...
    procedure DoSomething;
     ...
  end;
```

DoSomething 的定义声明必须在模块的后面出现：

```
procedure TMyClass.DoSomething;
begin
 ...
end;
```

虽然类声明既可以出现在单元的 interface 部分，也可以出现在 implementation 部分，但类方法的实现（定义声明）必须出现在 implementation 部分。

在定义声明的头部，方法名总是使用类名进行限定。在方法的头部可以重新列出类声明时的参数，若这样做的话，参数的顺序、类型以及名称必须完全相同，若方法是函数的话，返回值也必须相同。

方法声明可包含一些特殊指示字，而它们不会出现在其它函数或过程中。指示字应当只出现在类声明中，并且以下面的顺序列出：

reintroduce; **overload**; *binding*; *calling convention*; **abstract**; *warning*

这里，*binding* 是 **virtual**、**dynamic** 或 **override**；*calling convention* 是 **register**、**pascal**、**cdecl**、**stdcall** 或 **safecall**；*warning* 是 **platform**、**deprecated** 或 **library**。

Inherited（继承）

关键字 **inherited** 在实现多态行为时扮演着特殊角色，它出现在方法定义中，后面跟一个标志符或者不跟。若 **inherited** 后面跟一个成员名称，它表示一个通常的方法调用，或者是引用一个属性或字段（except that the search for the referenced member begins with the immediate ancestor of the enclosing method's class）。比如，当

```
inherited Create(...);
```

出现在方法定义中时，它调用继承的 Create 方法。

When inherited has no identifier after it, it refers to the inherited method with the same name as the enclosing method. In this case, inherited takes no explicit parameters, but passes to the inherited method the same parameters with which the enclosing method was called. For example,

当 **inherited** 后面没有标志符时，它指的是和当前方法同名的继承下来的方法。在这种情况下，**inherited** 没有明确指定参数，但把当前使用的参数传给继承下来的方法。比如，

```
inherited;
```

经常出现在构造函数的实现中，它把相同的参数传给继承下来的构造函数。

Self（Self 变量）

在实现方法时，标志符 **Self** 引用方法所属的对象。比如，下面是 Classes 单元中 TCollection 的 Add 方法的实现：

```
function TCollection.Add: TCollectionItem;
begin
  Result := FItemClass.Create(Self);
end;
```

Add 方法调用 FItemClass 的 Create 方法，而 FItemClass 所属的类总是 TCollectionItem 的派生类，TCollectionItem.Create 有一个 TCollection 类型的单一参数，所以，Add 把此时 TCollection 的实例传给它，这以下面的代码表示：

```
var MyCollection: TCollection;
 ...
MyCollection.Add     // MyCollection 被传给 TCollectionItem.Create 方法
```

Self 在很多方面都有用，比如，一个在类中声明的成员（标志符）可能在方法中被重新声明，这种情况下，你可以使用 Self.*Identifier* 来访问原来的成员。

关于类方法中的 Self，请参考 Class methods。

Method binding（方法绑定）

Method binding: Overview（概述）

方法分为静态方法（默认）、虚方法和动态方法。虚方法和动态方法能被覆盖，它们可是是抽象的。当某个类类型的变量存储的是它的派生类时，它们的意义开始发挥作用，它们决定了调用方法时哪种实现被执行。

Static methods（静态方法）

方法默认是静态的。当调用一个静态方法时，类或对象被声明的类型决定了哪种实现被执行（编译时决定）。在下面的例子中，Draw 方法是静态的。

```
type
  TFigure = class
    procedure Draw;
  end;
  TRectangle = class(TFigure)
    procedure Draw;
  end;
```

给定上面的声明，下面的代码演示了静态方法执行时的结果。在第 2 个 Figure.Draw 中，变量 Figure 引用的是一个 TRectangle 类型的对象，但却执行 TFigure 中的 Draw 方法，因为 Figure 变量声明的类型是 TFigure。

```
var
  Figure: TFigure;
  Rectangle: TRectangle;
```

```
begin
  Figure := TFigure.Create;
  Figure.Draw;                    // 调用 TFigure.Draw
  Figure.Destroy;
  Figure := TRectangle.Create;
  Figure.Draw;                    // 调用 TFigure.Draw
  TRectangle(Figure).Draw;        // 调用 TRectangle.Draw
  Figure.Destroy;
  Rectangle := TRectangle.Create;
  Rectangle.Draw;                 // 调用 TRectangle.Draw
  Rectangle.Destroy;
end;
```

Virtual and dynamic methods（虚方法和动态方法）

要实现虚方法或动态方法，在声明时包含 **virtual** 或 **dynamic** 指示字。不像静态方法，虚方法和动态方法能在派生类中被覆盖。当调用一个被覆盖的方法时，类或对象的实际类型决定了哪种实现被调用（运行时），而不是它们被声明的类型。

要覆盖一个方法，使用 **override** 指示字重新声明它就可以了。声明被覆盖的方法时，它的参数的类型和顺序以及返回值（若有的话）必须和祖先类相同。

在下面的例子中，TFigure 中声明的 Draw 方法在它的两个派生类中被覆盖了。

```
type
  TFigure = class
    procedure Draw; virtual;
  end;
  TRectangle = class(TFigure)
    procedure Draw; override;
  end;
  TEllipse = class(TFigure)
    procedure Draw; override;
  end;
```

给定上面的声明，下面代码演示了虚方法被调用时的结果，在运行时，执行方法的变量，它的实际类型是变化的。

```
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  Figure.Draw;         // 调用 TRectangle.Draw
  Figure.Destroy;
  Figure := TEllipse.Create;
  Figure.Draw;         // 调用 TEllipse.Draw
  Figure.Destroy;
end;
```

只有虚方法和动态方法能被覆盖，但是，所有方法都能被重载，请参考 Overloading methods。

Virtual versus dynamic（比较虚方法和动态方法）

虚方法和动态方法在语义上是相同的，唯一的不同是在运行时决定方法调用的实现方式上，虚方法在速度上进行了优化，而动态方法在代码大小上做了优化。

通常情况下，虚方法是实现多态行为的最有效的实现方式。当基类声明了大量的要被许多派生类继承的（可覆盖的）方法、但只是偶尔才覆盖时，动态方法还是比较有用的。

Overriding versus hiding（比较覆盖和隐藏）

在声明方法时，如果它和继承的方法具有相同的名称和参数，但不包含 **override**，则新方法仅仅是隐藏了继承下来的方法，并没有覆盖它。这样，两个方法在派生类中都存在，方法名是静态绑定的。比如，

```
type
  T1 = class(TObject)
    procedure Act; virtual;
  end;
  T2 = class(T1)
    procedure Act;        // 重新声明 Act，但没有覆盖
  end;
var
  SomeObject: T1;
begin
  SomeObject := T2.Create;
  SomeObject.Act;         // 调用 T1.Act
end;
```

Reintroduce（重新引入）

reintroduce 指示字告诉编译器，当隐藏一个先前声明的虚方法时，不给出警告信息。比如，

```
procedure DoSomething; reintroduce;  // 父类也有一个 DoSomething 方法
```

当要使用新方法隐藏继承下来的虚方法时，使用 reintroduce 指示字。

Abstract methods（抽象方法）

抽象方法是虚方法或动态方法，并且在声明它的类中没有实现，而是由它的派生类来实现。声明抽象方法时，必须在 **virtual** 或 **dynamic** 后面使用 **abstract** 指示字。比如，

```
procedure DoSomething; virtual; abstract;
```

只有当抽象方法在一个类中被覆盖时，你才能使用这个类或它的实例进行调用。

Overloading methods（重载方法）

一个方法可以使用 **overload** 指示字来重新声明，此时，若重新声明的方法和祖先类的方法具有不同的参数，它只是重载了这个方法，并没有隐藏它。当在派生类中调用此方法时，依靠参数来决定到底调用哪一个。

若要重载一个虚方法，在派生类中重新声明时使用 **reintroduce** 指示字。比如，

```
  type
    T1 = class(TObject)
      procedure Test(I: Integer); overload; virtual;
    end;
    T2 = class(T1)
      procedure Test(S: string); reintroduce; overload;
    end;
     ...
    SomeObject := T2.Create;
    SomeObject.Test('Hello!');                // 调用 T2.Test
    SomeObject.Test(7);                       // 调用 T1.Test
```

在一个类中，你不能以相同的名字公布（**published**）多个重载的方法，维护 RTTI 信息要求每一个公布的成员具有不同的名字。

```
type
  TSomeClass = class
  published
    function Func(P: Integer): Integer;
    function Func(P: Boolean): Integer      // 错误
     ...
```

作为属性读写限定符的方法不能被重载。

实现重载的方法时，必须重复列出类声明时方法的参数列表。关于重载的更多信息，请参考 Overloading procedures and functions。

Constructors（构造函数）

构造函数是一个特殊的方法，用来创建和初始化一个实例对象。声明一个构造函数就像声明一个过程，但以 **constructor** 开头。比如：

```
  constructor Create;
  constructor Create(AOwner: TComponent);
```

构造函数必须使用默认的 **register** 调用约定。虽然声明中没有指定返回值，但构造函数返回它创建的对象引用，或者对它进行调用的对象（的引用）。

一个类的构造函数可以不止一个，但大部分只有一个。按惯例，构造函数通常命名为 Create。

要创建一个对象，在类（标志符）上调用构造函数。比如，

```
  MyObject := TMyClass.Create;
```

它在堆中为对象分配内存，并设置所有的有序类型的字段为 0，把 **nil** 赋给所有的指针和类类型的字段，使所有的字符串类型的字段为空；接下来，构造函数中指定的其它动作（命令）开始执行，通常，初始化对象是基于传给构造函数的参数值；最后，构造函数返回新创建的对象的引用，此时它已完成了初始化。返回值的类型与调用构造函数的类相同。

当使用类引用来调用构造函数时，若执行过程中发生了异常，则自动调用析构函数 *Destroy* 来销毁不完整的对象。

当使用对象引用来调用构造函数时（而不是使用类引用），它不是创建一个对象；取而代之的是，构造函数作用在指定的对象上，它只是执行构造函数中的命令语句，然后返回一个对象的引用（是怎样的对象引用，和调用它的一样吗？）。使用对象引用来调用构造函数时，通常和关键字 **inherited** 一起使用来调用一个继承的构造函数。

下面是一个类和构造函数的例子。

```
type
```

```
  TShape = class(TGraphicControl)
  private
    FPen: TPen;
    FBrush: TBrush;
    procedure PenChanged(Sender: TObject);
    procedure BrushChanged(Sender: TObject);
  public
    constructor Create(Owner: TComponent); override;
    destructor Destroy; override;
    ...
  end;
constructor TShape.Create(Owner: TComponent);
begin
  inherited Create(Owner);          // 初始化继承下来的部分
  Width := 65;                      // 改变继承下来的属性
  Height := 65;
  FPen := TPen.Create;              // 初始化新字段
  FPen.OnChange := PenChanged;
  FBrush := TBrush.Create;
  FBrush.OnChange := BrushChanged;
end;
```

构造函数的第一步，通常是调用继承下来的构造函数，对继承的字段进行初始化；然后对派生类中新引入的字段进行初始化。因为构造函数总是把为新对象分配的内存进行"清零"（clear），所以，对象的所有字段开始时都是 0（有序类型）、**nil**（指针和类）、空（字符串）或者 **Unassigned**（变体类型）。所以，除非字段的值不为 0 或者空值，我们没必要在构造函数中初始化各字段。

当使用类标志符调用构造函数时，声明为虚方法的构造函数和声明为静态时是相同的。但是，当和类引用（class-reference）结合使用时，虚构造函数允许使用多态，也就是说，在编译时，对象的类型是未知的（参考 Class references）。

Destructors（析构函数）

析构函数是一个特殊的方法，用来销毁调用的对象并且释放它的内存。声明一个析构函数就像声明一个过程，但以 **destructor** 开头。比如：

```
  destructor Destroy;
  destructor Destroy; override;
```

析构函数必须使用默认的 **register** 调用约定。虽然一个类的析构函数可以不止一个，但推荐每个类覆盖继承下来的 **Destroy** 方法，并不再声明其它析构函数。

要调用析构函数，必须使用一个实例对象的引用。比如，

```
  MyObject.Destroy;
```

当析构函数被调用时，它里面的命令首先被执行，通常，这包括销毁所有的嵌入对象以及释放为对象分配的资源；接下来，为对象分配的内存被清除。

下面是一个析构函数实现的例子：

```
  destructor TShape.Destroy;
  begin
    FBrush.Free;
    FPen.Free;
```

```
    inherited Destroy;
  end;
```

析构函数的最后一步，通常是调用继承下来的析构函数，用来销毁继承的字段。

When an exception is raised during creation of an object, Destroy is automatically called to dispose of the unfinished object. This means that Destroy must be prepared to dispose of partially constructed objects. Because a constructor sets the fields of a new object to zero or empty values before performing other actions, class-type and pointer-type fields in a partially constructed object are always nil. A destructor should therefore check for nil values before operating on class-type or pointer-type fields. Calling the Free method (defined in TObject), rather than Destroy, offers a convenient way of checking for nil values before destroying an object.

当创建对象时发生了异常，会自动调用析构函数来清除不完整的对象，这表示析构函数必须准备好来清除只构建了一部分的对象。因为构造函数在执行其它动作之前先设置新对象的字段为 0 或空值，在一个只构建了一部分的对象中，类类型和指针类型的字段总是 **nil**，所以，在操作类类型和指针类型的字段时，析构函数必须检查它们是否为 **nil**。销毁一个对象时调用 Free 方法（在 TObject 中定义）而不是 Destroy 会更加方便，因为前者会检查对象是否为 **nil**。

Message methods（Message 方法）

Message 方法用来响应动态分派的消息。Message 方法在各个平台上都是支持的，VCL 使用 message 方法来响应 Windows 消息，CLX 不使用 message 方法来响应系统事件。

在声明方法时，通过包含 **message** 指示字来创建一个 message 方法，并在 **message** 后面跟一个介于 1 到 49151 之间的整数常量，它指定消息的号码（ID）。对于 VCL 控件（control），message 方法中的整数常量可以是 Messages 单元中定义的 Windows 消息号码，这里还定义了相应的记录类型。一个 message 方法必须是具有一个单一 **var** 参数的过程。

比如，在 Windows 下：

```
  type
    TTextBox = class(TCustomControl)
    private
      procedure WMChar(var Message: TWMChar); message WM_CHAR;
      ...
    end;
```

比如，在 Linux 或在跨平台的情况下，你要以如下方式处理消息：

```
  const
    ID_REFRESH = $0001;
  type
    TTextBox = class(TCustomControl)
    private
      procedure Refresh(var Message: TMessageRecordType); message ID_REFRESH;
      ...
    end;
```

Message 方法不必包含 **override** 指示字来覆盖一个继承的 message 方法。实际上，在覆盖方法时也不必指定相同的方法名称和参数类型，而只要一个消息号码就决定了这个方法响应哪个消息和是否覆盖一个方法。

Implementing message methods（实现 message 方法）

The implementation of a message method can call the inherited message method, as in this example (for Windows):

实现一个 message 方法时，可以调用继承的 message 方法，就像下面的例子（适用于 Windows）：

```
procedure TTextBox.WMChar(var Message: TWMChar);
begin
  if Chr(Message.CharCode) = #13 then
    ProcessEnter
  else
    inherited;
end;
```

在 Linux 或跨平台的情况下，你要以如下方式实现同样的目的：

```
procedure TTextBox.Refresh(var Message: TMessageRecordType);
begin
  if Chr(Message.Code) = #13 then
    ...
  else
    inherited;
end;
```

命令 **inherited** 按类的层次结构向后寻找，它将调用和当前方法具有相同消息号码的第一个（message）方法，并把消息记录（参数）自动传给它。如果没有祖先类实现 message 方法来响应给定的消息号码，**inherited** 调用 TObject 的 DefaultHandler 方法。

DefaultHandler 没有做任何事，只是简单地返回而已。通过覆盖 DefaultHandler，一个类可以实现自己对消息的响应。在 Windows 下，VCL 控件（control）的 DefaultHandler 方法调用 Windows 的 DefWindowProc（API）函数。

Message dispatching（消息分派）

消息处理函数很少直接调用，相反，消息是通过继承自 TObject 的 Dispatch 方法来分派给对象的。

```
procedure Dispatch(var Message);
```

传给 Dispatch 的参数 Message 必须是一个记录，并且它的第一个字段是 Cardinal 类型，用来存储消息号码。

Dispatch 按类的层次结构向后搜索（从调用对象所属的类开始），它将调用和传给它的消息具有相同号码的 message 方法。若没有发现指定号码的 message 方法，Dispatch 调用 DefaultHandler。

Properties（属性）

Properties: Overview（概述）

属性就像一个字段，它定义了对象的一个特征。但字段仅仅是一个存储位置，它的内容可以被查看和修改；而属性通过读取或修改它的值与特定的行为关联起来。属性通过操纵一个对象的特征来提供对它的控制，它们还使特征能被计算。

声明属性时要指定名称和类型，并且至少包含一个访问限定符。属性声明的语法是

```
property propertyName[indexes]: type index integerConstant specifiers;
```

这里

- *propertyName* 是任何有效标志符；
- [*indexes*]是可选的，它是用分号隔开的参数声明序列，每个参数声明具有如下形式：*identifier*1, ..., *identifier*n: *type*。更多信息请参考 Array properties；
- *type* 必须是内置的或前面声明的数据类型，也就是说，像 **property** Num: 0..9 ...这样的属性声明是非法的；

- index *integerConstant* 子句是可选的。更多信息请参考 Index specifiers；
- specifiers 是由 **read**、**write**、**stored**、**default**（或 **nodefault**）和 **implements** 限定符组成的序列。每个属性声明必须至少包含一个 **read** 或 **write** 限定符。

属性由它们的访问限定符定义。不像字段，属性不能作为 **var** 参数传递，也不能使用@运算符，原因是属性不一定（是不一定，还是一定不呢？）在内存中存在。比如，它可能有一个读方法从数据库中检索一个值或者产生一个随机数值。

Property access（属性访问）

每个属性有一个读限定符，一个写限定符，或两者都有，它们称为访问限定符，具有以下的格式

read *fieldOrMethod*
write *fieldOrMethod*

这里，*fieldOrMethod* 是一个字段或方法名，它们既可以和属性在同一个类中声明，也可以在祖先类中声明。

- 如果 *fieldOrMethod* 和属性是在同一个类中声明的，它必须出现在属性声明的前面；如果它是在祖先类中声明的，则它对派生类必须是可见的，也就是说，若祖先类在不同的单元声明，则 *fieldOrMethod* 不能是私有的字段或方法；
- 若 *fieldOrMethod* 是一个字段，它的类型和属性必须相同；
- 若 *fieldOrMethod* 是一个方法，它不能是重载的，而且，对于公布的属性，访问方法必须使用默认的 **register** 调用约定；
- 在读限定符中，若 *fieldOrMethod* 是一个方法，它必须是一个不带参数的函数，并且返回值和属性具有相同的类型；
- 在写限定符中，若 *fieldOrMethod* 是一个方法，它必须是一个带有单一值参（传值）或常量参数的过程，这个参数和属性具有相同的类型；

比如，给定下面的声明

```
property Color: TColor read GetColor write SetColor;
```

GetColor 方法必须被声明为：

```
function GetColor: TColor;
```

SetColor 方法必须被声明为下面之一：

```
procedure SetColor(Value: TColor);
procedure SetColor(const Value: TColor);
```

（当然，SetColor 的参数名不必非得是 Value。）

当在表达式中使用属性时，通过在读限定符中列出的字段或方法读取它的值；当在赋值语句中使用属性时，通过写限定符列出的字段或方法对它进行写入。

在下面的例子中，我们声明了一个叫做 TCompass 的类，它有一个公布的属性 Heading。Heading 的值通过 FHeading 字段读取，写入时使用 SetHeading 过程。

```
type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    property Heading: THeading read FHeading write SetHeading;
    …
  end;
```

给出上面的声明，语句

```
if Compass.Heading = 180 then GoingSouth;
Compass.Heading := 135;
```

对应于

```
if Compass.FHeading = 180 then GoingSouth;
Compass.SetHeading(135);
```

在 TCompass 类中，读取 Heading 属性时没有执行任何命令，只是取回存储在 FHeading 字段的值；另一方面，给 Heading 赋值变成了对 SetHeading 方法的调用，我们推测，它的操作将是把新值存储在 FHeading 字段，还可能包括其它命令。比如，SetHeading 可能以如下方式实现：

```
procedure TCompass.SetHeading(Value: THeading);
begin
  if FHeading <> Value then
  begin
    FHeading := Value;
    Repaint;            // 刷新用户界面来反映新值
  end;
end;
```

若声明属性时只有读限定符，则它是只读属性；若只有写限定符，则它是只写属性。当给一个只读属性赋值，或在表达式中使用只写属性时都将产生错误。

Array properties（数组属性）

数组属性是被索引的属性，它们能表示像下面的一些事物：列表中的条目、一个控件的子控件和位图中的象素等等。

声明数组属性时包含一个参数列表，它指定索引的名称和类型，比如，

```
property Objects[Index: Integer]: TObject read GetObject write SetObject;
property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
property Values[const Name: string]: string read GetValue write SetValue;
```

索引参数列表的格式和过程（或函数）的参数列表相同，除了使用中括号取代了圆括号。不像数组只使用有序类型的索引，数组属性的索引能使用任何类型。

对数组属性，访问限定符必须使用方法而不是字段。读限定符的方法必须是一个函数，它的参数数目、类型以及顺序必须和索引中列出的一致，并且返回值和属性是同一类型；对写限定符，它必须是一个过程，这个过程必须使用索引中列出的参数，包括数目、类型以及顺序必须相同，另外再加一个和属性具有相同类型的值参（传值）或常量参数。

比如，前面的属性可能具有如下的访问方法声明：

```
function GetObject(Index: Integer): TObject;
function GetPixel(X, Y: Integer): TColor;
function GetValue(const Name: string): string;
procedure SetObject(Index: Integer; Value: TObject);
procedure SetPixel(X, Y: Integer; Value: TColor);
procedure SetValue(const Name, Value: string);
```

一个数组属性通过使用属性索引来进行访问。比如，语句

```
if Collection.Objects[0] = nil then Exit;
Canvas.Pixels[10, 20] := clRed;
Params.Values['PATH'] := 'C:\DELPHI\BIN';
```

对应于

```
if Collection.GetObject(0) = nil then Exit;
Canvas.SetPixel(10, 20, clRed);
Params.SetValue('PATH', 'C:\DELPHI\BIN');
```

在 Linux 下，上面的例子你要使用像“/usr/local/bin”的路径取代“C:\DELPHI\BIN”。

定义数组属性时可以在后面使用 **default** 指示字，此时，数组属性变成类的默认属性。比如，

```
type
  TStringArray = class
  public
    property Strings[Index: Integer]: string ...; default;
    ...
  end;
```

若一个类有默认属性，你能使用缩写词 *object*[*index*]来访问这个属性，它就相当于 *object.property*[*index*]。比如，给定上面的声明，StringArray.Strings[7]可以缩写为 StringArray[7]。一个类只能有一个默认属性，在派生类中改变或隐藏默认属性可能导致无法预知的行为，因为编译器总是静态绑定一个对象地默认属性。

Index specifiers（索引限定符）

索引限定符能使几个属性共用同一个访问方法来表示不同的值。索引限定符包含 **index** 指示字，并在后面跟一个介于-2147483647 到 2147483647 之间的整数常量。若一个属性有索引限定符，它的读写限定符必须是方法而不能是字段。比如，

```
type
  TRectangle = class
  private
    FCoordinates: array[0..3] of Longint;
    function GetCoordinate(Index: Integer): Longint;
    procedure SetCoordinate(Index: Integer; Value: Longint);
  public
    property Left: Longint index 0 read GetCoordinate write SetCoordinate;
    property Top: Longint index 1 read GetCoordinate write SetCoordinate;
    property Right: Longint index 2 read GetCoordinate write SetCoordinate;
    property Bottom: Longint index 3 read GetCoordinate write SetCoordinate;
    property Coordinates[Index: Integer]: Longint read GetCoordinate write
SetCoordinate;
    ...
  end;
```

对于有索引限定符的属性，它的访问方法必须有一个额外的整数类型的值参：对于读取函数，它必须是最后一个参数；对于写入过程，它必须是倒数第 2 个参数（在指定属性值的参数之前）。当程序访问属性时，属性的整数常量自动传给访问方法。

给出上面的声明，若 Rectangle 属于 TRectangle 类型，则

```
Rectangle.Right := Rectangle.Left + 100;
```

对应于

```
Rectangle.SetCoordinate(2, Rectangle.GetCoordinate(0) + 100);
```

Storage specifiers（存储限定符）

可选指示字 **stored**、**default** 和 **nodefault** 被称为存储限定符，它们对程序的行为没有影响，但决定了 RTTI 的维护方式，它们决定是否把公布属性的值存储到窗体文件中。

stored 指示字后面必须跟 True、False、Boolean 类型的字段名或者一个返回 Boolean 值的无参数方法。比如，

```
property Name: TComponentName read FName write SetName stored False;
```

若一个属性没有 **stored** 指示字，就相当于指定了 **stored** True。

default 指示字后面必须跟随一个和属性具有相同类型的常量，比如，

```
property Tag: Longint read FTag write FTag default 0;
```

要覆盖一个继承下来的默认值而不指定新值，使用 **nodefault** 指示字。**default** 和 **nodefault** 只支持有序类型和集合类型（当它的基础类型是有序类型，并且上下边界都在 0 到 31 之间时）。若声明属性时没有使用 **default** 或者 **nodefault**，它被当作 **nodefault** 看待。对于实数、指针和字符串，它们分别有隐含的默认值 0、**nil** 和 ''（空串）

当保存一个组件的状态时，组件中公布属性的存储限定符会被检查，若属性的当前值和默认值不同（或没有默认值），并且 **stored** 为 True，则它的值就会被保存；否则，属性的值不被保存。

注意：存储限定符不支持数组属性。在声明数组属性时，指示字 **default** 有不同的意义。

Property overrides and redeclarations（属性的覆盖和重新声明）

声明时没有指定类型的属性称为属性覆盖，它允许你改变一个属性继承下来的可见性或限定符。最简单的覆盖只包含关键字 **property**、并在后面跟属性标志符，这种方式用来改变属性的可见性。比如，祖先类声明了一个受保护的属性，派生类可以重新声明它为公有的或公布的。属性覆盖可包含 **read**、**write**、**stored**、**default** 和 **nodefault**，它们覆盖了继承下来的相应指示字。覆盖可以取代访问限定符、添加限定符或增大属性的可见性，但不能删除访问限定符或降低可见性。覆盖可包含 **implements** 指示字，它添加可以实现的接口，但不能删除继承下来的那些。

下面的声明演示了属性覆盖的使用：

```
type
  TAncestor = class
   …
  protected
    property Size: Integer read FSize;
    property Text: string read GetText write SetText;
    property Color: TColor read FColor write SetColor stored False;
    …
  end;
type
  TDerived = class(TAncestor)
   …
  protected
    property Size write SetSize;
  published
    property Text;
    property Color stored True default clBlue;
    …
```

```
    end;
```

覆盖的 Size 属性添加了写限定符，允许属性能被修改；覆盖的 Text 和 Color 属性把可见性从 **protected** 改变为 **published**；覆盖的 Color 属性还指定若它的值不为 clBlue，它将被保存进文件。

若重新声明属性时包含类型标志符，这将隐藏继承下来的属性而不是覆盖它，也就是创建了一个（和继承下来的属性）具有相同名称的新属性。任何指定类型的属性声明必须是完整的，也就至少要包含一个访问限定符。

派生类中属性是隐藏还是覆盖呢？属性的查找总是静态的，也就是说，对象（变量）声明的类型决定了它的属性。所以，在下面的代码执行后，读取 MyObject.Value 或给它赋值将调用 Method1 或 Method2，即使 MyObject 存储的是 TDescendant 的一个实例；但你可以把 MyObject 转换为 TDescendant 来访问派生类的属性和它们的访问限定符。

```
type
  TAncestor = class
     …
    property Value: Integer read Method1 write Method2;
  end;
  TDescendant = class(TAncestor)
     …
    property Value: Integer read Method3 write Method4;
  end;
var MyObject: TAncestor;
 …
MyObject := TDescendant.Create;
```

Class references（类引用）

Class references: Overview（概述）

有时，我们需要使用类本身而不是它的实例（也就是对象），比如，当使用类引用来调用构造函数时。你总是能使用类名来引用一个类，但有时，你也需要声明变量或参数把类作为它的值，这种情况下，你需要使用类引用类型。

Class-reference types（类引用类型）

类引用类型有时称为元类，用如下的构造形式表示

```
class of type
```

这里，*type* 是任何类类型。*type*（标志符）本身表示一个 class of *type*（元类）类型的值。若 type1 是 type2 的祖先类，则 class of type2（元类）和 class of type1（元类）是赋值兼容的。这样

```
type TClass = class of TObject;
var AnyObj: TClass;
```

声明了一个叫做 AnyObj 的变量，它能存储任何类引用。类引用类型的声明不能直接用于变量或参数声明中。你能把 **nil** 值赋给任何类引用变量。

要了解类引用类型如何使用，看一下 TCollection（在 Classes 单元）的构造函数声明：

```
type TCollectionItemClass = class of TCollectionItem;
  …
```

```
constructor Create(ItemClass: TCollectionItemClass);
```

上面声明说，要创建一个 TCollection 实例对象，你必须向构造函数传递一个类名，它属于 TCollectionItem 类或是它的派生类。

当你调用一个类方法，或者调用一个类（或对象）的虚构造函数（编译时它们的类型不能确定）时，类引用是很有用的。

Constructors and class references（构造函数和类引用）

构造函数可通过一个类引用类型的变量进行调用，这允许创建编译时类型并不确定的对象。比如，

```
type TControlClass = class of TControl;
function CreateControl(ControlClass: TControlClass;
  const ControlName: string; X, Y, W, H: Integer): TControl;
begin
  Result := ControlClass.Create(MainForm);
  with Result do
  begin
    Parent := MainForm;
    Name := ControlName;
    SetBounds(X, Y, W, H);
    Visible := True;
  end;
end;
```

CreateControl 函数需要一个类引用类型的参数，它指定创建何种控件，函数使用这个参数来调用构造函数。因为类标志符（类名）表示一个类引用的值，所以能使用它作为参数来调用 CreateControl 创建一个实例。比如，

```
CreateControl(TEdit, 'Edit1', 10, 10, 100, 20);
```

使用类引用来调用的构造函数通常是虚方法，实际调用的构造函数（指实现）由运行时类引用的类型决定。

Class operators（类运算符）

Class operators: Overview（概述）

每个类从 TObject 继承了两个分别叫做 ClassType 和 ClassParent 的方法，前者返回对象的类引用，后者返回对象的父类类引用。这两个方法的返回值都是 TClass（这里 TClass = class of TObject）类型，它们能被转换为更加明确的类型。每个类还继承了一个叫做 InheritsFrom 的方法，它测试调用的对象是否从一个指定的类派生而来（如果对象是类的一个实例，结果如何？）。这些方法被 **is** 和 **as** 运算符使用，很少直接调用它们。

The is operator（is 运算符）

is 运算符执行动态类型检查，用来验证运行时一个对象的实际类型。

```
object is class
```

若 *object* 对象是 *class* 类的一个实例，或者是 *class* 派生类的一个实例，上面的表达式返回 True，否则返回 False（若 *object* 是 **nil**，则结果为 False）。如果 *object* 声明的类型和 *class* 不相关，也就是说，若两个类不同并且其中一个不是另一个的祖先，则发生编译错误。比如，

```
if ActiveControl is TEdit then TEdit(ActiveControl).SelectAll;
```

上面的语句先检查一个对象（变量）是否是 TEdit 或它的派生类的一个实例，然后再决定是否把它转换为 TEdit。

The as operator（as 运算符）

as 运算符执行受检查的类型转换。表达式

```
object as class
```

返回和 *object* 相同的对象引用，但它的类类型是 *class*。在运行时，*object* 对象必须是 *class* 类的一个实例，或者是它的派生类的一个实例，或者是 **nil**，否则将产生异常；若 object 声明的类型和 class 不相关，也就是说，若两个类不同并且其中一个不是另一个的祖先，则发生编译错误。比如，

```
with Sender as TButton do
begin
  Caption := '&Ok';
  OnClick := OkClick;
end;
```

因为运算符优先权的问题，我们经常需要把 **as** 类型转换放在一对括号中，比如，

```
(Sender as TButton).Caption := '&Ok';
```

Class methods（类方法）

类方法是作用在类而不是对象上面的方法（不同于构造函数）。类方法的定义必须以关键字 **class** 开始，比如，

```
type
  TFigure = class
  public
    class function Supports(Operation: string): Boolean; virtual;
    class procedure GetInfo(var Info: TFigureInfo); virtual;
    ...
  end;
```

类方法的定义部分也必须以 **class** 开始，比如，

```
class procedure TFigure.GetInfo(var Info: TFigureInfo);
begin
  ...
end;
```

在类方法的定义部分，**Self** 表示调用方法的类（which could be a descendant of the class in which it is defined，它或许是定义方法的类的一个派生类）。若使用类 C 调用方法，Self 的类型是 class of C（元类）。所以，你不能使用 Self 访问字段、属性和平常的方法（由对象调用的方法），但能调用构造函数和其它类方法。

类方法既可以通过类引用来调用，也可以使用对象，当使用后者时， **Self** 值等于对象所属的类。

Exceptions（异常）

Exceptions: Overview（概述）

当发生错误或其它事件而打断了程序的正常执行时，将引发一个异常。异常把控制权交给一个异常处理程序（exception handler），这使我们把错误处理和正常的程序逻辑隔离开来。因为异常属于对象，我们可以应用继承关系把它们分层组织，在不影响现有代码的情况下能引入新的异常。异常能传送一些信息（比如错误消息），把它们从异常发生点带到被处理的地方。

当程序使用 SysUtils 单元时，所有的运行时错误都将被转换为异常，否则，像内存不足、被零除、GPF（general protection fault）等错误会终止程序，而现在它们能被捕获并进行处理。

When to use exceptions（何时使用异常）

异常提供了一种优雅的方式来捕获运行时错误，而不是挂起程序和使用笨拙的条件语句。但是，Object Pascal 异常处理机制的复杂性降低了它的效率，所以应当酌情使用。虽然（几乎）能以任何原因引发一个异常，也可以把（几乎）任何代码段使用 **try...except** 或 **try...finally** 封装起来进行保护，但实际上最好把它们用在特殊情况。

异常处理程序适用于以下几种情况：发生几率比较低或难以预料、但结果却是灾难性（比如程序崩溃）的错误；对于 **if...then** 语句来说，错误条件非常复杂或难以判断；当需要响应操作系统引发的异常，或一些你不能得到源码而又必须对它们的异常做出响应的例程。异常通常用在硬件、内存、I/O 和操作系统错误。

条件语句经常是判断错误的最好方式。比如，假设你要在打开一个文件之前先确定它是否存在，你以下面的方式实现它：

```
try
  AssignFile(F, FileName);
  Reset(F);              // 若没有发现文件则引发一个 EinOutError 异常
except
  on Exception do ...
end;
```

但你也可以使用下面的方式来避免异常处理的开销

```
if FileExists(FileName) then       // 若没有发现文件则返回 False，不会引发异常
begin
  AssignFile(F, FileName);
  Reset(F);
end;
```

Assertions 提供了另一种方式，使你在源代码的任何地方判断一个布尔条件。当一个 Assert 语句失败时，程序或者挂起，或者引发一个 EAssertionFailed 异常（若它使用 SysUtils 单元的话）。只有当判断一个你不期望发生的条件时，你才应该使用 Assertions。要了解更多信息，请参考在线帮助中的 the standard procedure Assert。

Declaring exception types（声明异常类型）

异常类的声明和其它类一样，实际上，使用任何类的一个实例表示异常是可行的，但推荐从 SysUtils 单

元的 Exception 类进行派生。

你能应用继承关系给异常分组，比如，下面是 SysUtils 单元中的声明，它为计算错误定义了一组异常类

```
type
  EMathError = class(Exception);
  EInvalidOp = class(EMathError);
  EZeroDivide = class(EMathError);
  EOverflow = class(EMathError);
  EUnderflow = class(EMathError);
```

给定上面的声明，你能定义一个单一的 EMathError 异常处理程序，它也能处理 EInvalidOp、EZeroDivide、EOverflow 和 EUnderflow 异常。

有时，异常类会定义字段、方法和属性，它们用来传达一些额外的错误信息。比如，

```
type EInOutError = class(Exception)
    ErrorCode: Integer;
  end;
```

Raising and handling exceptions（引发和处理异常）

Raising and handling exceptions（引发和处理异常）

要创建一个异常对象，在 **raise** 语句中调用异常类的构造函数。比如，

```
raise EMathError.Create;
```

通常，raise 语句的格式是

```
raise object at address
```

这里，*object* 和 at *address* 都是可选的。若省略了 object，则语句重新引发当前异常，请参考 Re-raising exceptions；当指定了一个地址，它通常是指向过程或函数的指针，使用这个选项，可使异常从堆栈中一个较早点引发，而不是从它实际发生的地点引发（use this option to raise the exception from an earlier point in the stack than the one where the error actually occurred）。

当引发一个异常时，也就是使用了 **raise** 语句（referenced in a raise statement），它将受异常处理逻辑的控制。一个 raise 语句永远不会以正常方式返回控制，相反，它把控制权交给能处理指定的异常（类）、并且在最内层的异常处理程序（最内层是指最后进入但还没有退出的一个 **try...except** 块）。

比如，下面的函数把一个字符串转换为整数，若结果超出指定的范围则引发一个 ERangeError 异常。

```
function StrToIntRange(const S: string; Min, Max: Longint): Longint;
begin
  Result := StrToInt(S);          // StrToInt 在 SysUtils 单元声明
  if (Result < Min) or (Result > Max) then
    raise ERangeError.CreateFmt(
      '%d is not within the valid range of %d..%d',
      [Result, Min, Max]);
end;
```

注意 **raise** 语句中调用的 CreateFmt 方法。Exception 和它的派生类有特殊的构造函数，提供了可选择的方法来创建异常消息和 context ID。

被引发的（raised）异常在处理后自动清除，永远不要试图手动销毁它。

注意：在单元的初始化部分引发一个异常可能无法产生预期的结果。对异常的正规支持来自 SysUtils 单元，在获得这种支持之前，它必须被初始化。如果在初始化期间产生了异常，所有被初始化的单元（包括 SysUtils）执行结束化处理，并重新引发异常。然后，通常是结束程序来捕获和处理异常（Then the

exception is caught and handled, usually by interrupting the program）。

Try … except statements（Try … except 语句）

异常在 **try...except** 语句中被处理，比如，

```
try
  X := Y/Z;
except
  on EZeroDivide do HandleZeroDivide;
end;
```

上面的语句尝试 Y 被 Z 除，若 EZeroDivide 异常发生，则调用例程 HandleZeroDivide。

try...except 语句的语法是

try *statements* **except** *exceptionBlock* **end**

这里，*statements* 是语句序列（由分号隔开的一系列语句），*exceptionBlock* 或者是

- 其它语句序列，或者是
- 一系列异常处理程序，后面跟可选的

  else *statements*

一个异常处理程序具有如下格式

on *identifier*: *type* do *statement*

这里，*identifier*:是可选的（若有的话，它可以是任何有效标志符），*type* 用来表示异常类，*statement* 是任何语句。

一个 **try...except** 语句执行开始处的（初始）代码，若没有引发异常，异常代码段（*exceptionBlock*）被忽略，程序控制转到下一部分。

若执行初始代码时发生了异常（或者执行了 raise 语句，或者是调用过程或函数引起的），都将试图对它进行处理：

- 若异常处理块（exception block）中有对应的异常，则控制权交给第一个匹配的处理程序。当处理程序中指定的异常类和（发生的）异常所属的类相同，或者是异常的祖先类时，我们说，这个异常处理程序与这个异常相“匹配”。
- 若没有发现相应的异常处理程序，当有 **else** 子句时，程序控制转到 **else** 子句。
- 若异常处理块中没有异常处理程序，而只是语句序列，则程序控制转到它的第一个语句。

如果上面的条件都不成立，会继续搜索下一个 **try...except** 语句块；若还没有合适的异常处理程序、或 **else** 子句或语句序列，搜索会继续扩展到下一个 **try...except** 语句块，依此类推。如果达到最外层的 **try...except** 语句块并且异常还没有被处理，程序就会终止。

当处理一个异常时，堆栈退回到包含 **try...except** 语句的过程或函数，程序控制权转给异常处理程序、else 子句或语句序列。这个过程忽略所有进入 **try...except** 后调用的过程和函数，然后，异常对象自动调用析构函数进行销毁，程序控制权转给 **try...except** 后面的语句。（如果调用 Exit、Break 或 Continue 使程序控制权离开了异常处理程序，异常对象也会自动销毁。）

在下面的例子中，第 1 个异常处理程序处理被 0 除异常，第 2 个处理溢出，最后一个处理其它的数学运算异常。EMathError 在最后出现，因为它是另外两个异常的祖先，若它最先出现，另外两个将永远不会被调用。

```
try
  …
except
  on EZeroDivide do HandleZeroDivide;
  on EOverflow do HandleOverflow;
```

```
    on EMathError do HandleMathError;
  end;
```

在异常处理程序中，可以在异常类之前指定一个标志符，在执行 **on...do** 后面的语句时，它表示异常对象，标志符的作用域被限定在这个语句中。比如，

```
  try
   ...
  except
    on E: Exception do ErrorDialog(E.Message, E.HelpContext);
  end;
```

若在异常处理块中使用了 **else** 子句，则它处理所有未经异常处理程序处理的异常。比如，

```
  try
   ...
  except
    on EZeroDivide do HandleZeroDivide;
    on EOverflow do HandleOverflow;
    on EMathError do HandleMathError;
  else
    HandleAllOthers;
  end;
```

这里，**else** 子句处理所有不是 EMathError 的异常。

若异常处理块没有异常处理程序，而只是包含一系列语句，则它们处理所有的异常。比如，

```
  try
   ...
  except
    HandleException;
  end;
```

这里，**try** 和 **except** 之间的代码在运行时产生的异常，都由 HandleException 例程进行处理。

Re-raising exceptions（重新引发一个异常）

当关键字 **raise** 在异常块中出现，并且它的后面没有对象引用时，它引发正在处理的异常。这使得异常处理程序能对错误做有限处理后重新引发它。对于发生异常后必须进行清除工作、但又不能进行全面处理的过程或函数，重新引发一个异常是有用的。

比如，GetFileList 函数分配一个 TStringList 对象，并用指定搜索路径下的文件名来填充它。

```
  function GetFileList(const Path: string): TStringList;
  var
    I: Integer;
    SearchRec: TSearchRec;
  begin
    Result := TStringList.Create;
    try
      I := FindFirst(Path, 0, SearchRec);
      while I = 0 do
      begin
        Result.Add(SearchRec.Name);
```

```
        I := FindNext(SearchRec);
      end;
    except
      Result.Free;
      raise;
    end;
  end;
```

GetFileList 创建一个 TStringList 对象，然后使用 FindFirst 和 FindNext 函数（在 SysUtils 单元定义）来初始化它。如果初始化失败（比如搜索路径无效，或者没有足够的内存来填充字符串列表），GetFileList 需要释放字符串列表，因为函数的调用者还不知道它的存在。由于这个原因，初始化字符串列表在 **try...except** 语句中执行，若发生了异常，异常处理块释放字符串列表，然后重新引发这个异常。

Nested exceptions（嵌套的异常）

对于异常处理程序，它自己也可以引发和处理异常。只要这些异常也是在异常处理程序的内部被处理，它们并不影响原来的异常；但是，若它超越了异常处理程序，原来的异常就会丢失。下面的 Tan 函数说明了这一点。

```
  type
    ETrigError = class(EMathError);
  function Tan(X: Extended): Extended;
  begin
    try
      Result := Sin(X) / Cos(X);
    except
      on EMathError do
        raise ETrigError.Create('Invalid argument to Tan');
    end;
  end;
```

若 Tan 在执行过程中发生了 EMathError 异常，则异常处理程序引发一个 ETrigError 异常。因为 Tan 没有为 ETrigError 提异常供处理程序，异常就传播到原异常处理程序的外面，从而导致 EMathError 被销毁。对于函数调用者来说，就像 Tan 函数引发了一个 ETrigError 异常。（不明白）

Try … finally statements（Try … finally 语句）

有时，我们希望不管有没有发生异常，指定的一部分操作都要被完全执行。比如，当一个例程需要控制一个资源，不管例程是否正常结束，能释放资源是非常重要的。在这种情况下，你可以使用 try...finally 语句。

下面的例子演示这段代码如何打开和处理一个文件，并且，即使在执行过程中发生了错误也能保证在最后关闭文件。

```
  Reset(F);
  try
   ...                // 处理文件 F
  finally
    CloseFile(F);
  end;
```

try...finally 语句的语法是

try *statementList*1 finally *statementList*2 end

这里，每个 *statementList* 是一系列由分号隔开的语句。**try...finally** 语句执行 *statementList*1（**try** 子句）中的命令，若它执行完毕并没有引发异常，*statementList*2（**finally** 子句）被执行。若在执行 *statementList*1 时发生了异常，程序控制权转给 *statementList*2，一旦它执行完毕，异常被重新引发。即使调用 Exit、Break 或 Continue 过程使程序控制权离开了 *statementList*1，*statementList*2 也会自动执行。所以，不论 **try** 子句如何结束，**finally** 子句总是被执行。

若异常发生了但却没有在 **finally** 子句中进行处理，异常会传播到 **try...finally** 语句的外面，这样，在 **try** 子句中已经引发的异常都会丢失。所以，**finally** 子句应当处理所有本地引发的异常，这样就不会打乱其它异常的传播。

Standard exception classes and routines（标准异常类和例程）

SysUtils 单元声明了几个标准例程来处理异常，它们包括 ExceptObject、ExceptAddr 以及 ShowException。SysUtils 和其它单元还包括很多异常类，它们（除了 OutlineError）都是从 Exception 派生而来。

Exception 类有 Message 和 HelpContext 的属性，它们用于传递错误描述和 context ID，后者用于上下文相关联机文档；它还定义了多个构造函数，使你能以不同的方式指定描述信息和 context ID。

Standard routines and I/O（标准例程和 I/O）

Standard routines and I/O: Overview（概述）

这些主题讨论文本和文件 I/O，并概述标准库例程。这里列出的很多过程和函数在 System 单元定义，此单元被隐含地编译到每个程序中；其它例程是内置于编译器的，就像它们在 System 单元一样。
一些标准例程在一些单元（比如 SysUtils）中，必须把这些单元列在 **uses** 子句中以使这些例程能被程序使用。但是，不能在 uses 子句中列出 System 单元，也不能修改 System 单元或试图重建它。

File input and output（文件输入和输出）

File input and output（文件输入和输出）

下面的表格列出了输入和输出例程。

过程或函数	描述
Append	打开一个已存在的文本文件进行追加
AssignFile	把一个外部文件的名称赋给文件变量
BlockRead	从一个无类型文件读取一个或多个记录
BlockWrite	向一个无类型文件写入一个或多个记录
ChDir	改变当前路径
CloseFile	关闭一个打开的文件
Eof	返回一个文件的 Eof 状态（判断是否在文件的末尾）
Eoln	返回一个文本文件的 Eoln 状态（判断是否在行的末尾）
Erase	删除一个外部文件
FilePos	返回一个类型或无类型文件的当前位置
FileSize	返回一个文件的当前大小，不能用于文本文件
Flush	清除一个输出文本文件的缓冲区（把它们的内容写入文件？）
GetDir	返回指定驱动器的当前路径
IOResult	返回一个整数值，它表示最后的 I/O 操作后的状态
MkDir	创建一个子目录
Read	从文件读取一个或多个值，把它们赋给一个或多个变量
Readln	和 Read 功能相同；对文本文件，它跳到下一行的开始
Rename	更改一个外部文件的名称
Reset	打开一个已存在的文件
Rewrite	创建和打开一个新文件
RmDir	删除一个空目录
Seek	把文件的当前位置移到指定的元素，适用于类型和无类型文件，不能用于文本文件
SeekEof	返回一个文本文件的 Eof 状态（判断是否在文件末尾）
SeekEoln	返回一个文本文件的 Eoln 状态（判断是否在行的末尾）
SetTextBuf	把 I/O 缓冲区赋给一个文本文件
Truncate	把一个类型或无类型文件在当前位置截断

Write	向一个文件写入一个或多个值
Writeln	和 Write 功能相同；对文本文件，它写入一个行结束（Eoln）标志

文件变量是任何类型为文件类型的变量。有三种类型的文件：类型文件、文本文件和无类型文件。声明文件类型的语法在 File types 一节中。

在使用文件变量前，必须调用 AssignFile 过程把它和一个外部文件相关联。外部文件通常是一个命名的磁盘文件，但它也可以是一个设备，比如键盘或显示器。外部文件存储写给文件的内容，或提供读取一个文件的内容。

一旦文件变量和外部文件完成关联，它必须被“打开”以便进行输入和输出操作。一个已存在的文件可使用 Reset 过程打开，使用 Rewrite 过程能创建一个新文件并打开它。使用 Reset 打开的文本文件是只读的，使用 Rewrite 和 Append 打开的文本文件只能写入。对类型文件和无类型文件，不管用 Reset 还是用 Rewrite 打开，它们都是可读写的。

每个文件是由元素（component）构成的线性序列，每个元素都有类型（或记录类型），元素从 0 开始计数。

通常，文件是被顺序访问的，也就是说，当使用 Read 读取一个元素，或使用 Write 写入一个元素时，当前文件位置移到下一个元素。对类型文件和无类型文件，能使用 Seek 进行随机访问，它把当前文件位置移到指定的元素。标准函数 FilePos 和 FileSize 能用来确定当前文件位置和当前文件大小。

当程序完成一个文件的处理时，必须使用 CloseFile 关闭文件。在文件关闭后，和它关联的外部文件被更新，文件变量就可以和其它外部文件关联了。

默认情况下，调用所有的 I/O 过程和函数会自动检查错误，如果发生错误，就引发一个异常（如果异常处理被禁止了，程序就结束）。这种自动检查可以通过编译器指示字**{$I+}**和**{$I−}**打开和关闭。当 I/O 检查被关闭，也就是说，当过程或函数调用是在**{$I−}**状态下被编译时，I/O 错误不会导致异常发生。要检查一个 I/O 操作的结果，你必须调用函数 IOResult。

你必须调用 IOResult 函数来清除错误，即使你对错误不感兴趣。如果你没有清除错误，并且当前状态是**{$I+}**，下一个 I/O 函数调用会因为遗留的错误而失败。

Text files（文本文件）

这一节对标准类型的文本文件进行概述。

当打开一个文本文件时，外部文件以一种特殊的方式被处理：它被看作是一系列的字符，这些字符被格式化为行，每行以一个 **Eoln** 标志（一个回车符，或许还跟一个换行符）结束。类型 Text 和 file of Char 不同。

对文本文件，由特殊形式的 Read 和 Write 读取和写入它们的值，这些值并不是 Char 类型，它们能自动转换成它们的字符表达形式。比如，Read(F, I)，这里 I 是一个整数变量，它读取一系列数字，并把它们解释为 10 进制整数，然后存储到 I 中。

There are two standard text-file variables, Input and Output. The standard file variable Input is a read-only file associated with the operating system’s standard input (typically, the keyboard). The standard file variable Output is a write-only file associated with the operating system’s standard output (typically, the display). Before an application begins executing, Input and Output are automatically opened, as if the following statements were executed:

有两个标准文本文件变量，Input 和 Output。Input 是一个只读文件，和操作系统的标准输入（通常是键盘）相关联。Output 是只写文件，和操作系统的标准输出（通常是显示器）相关联。在程序执行前，Input 和 Output 自动打开，就像下面的命令被执行。

```
AssignFile(Input, '');
Reset(Input);
AssignFile(Output, '');
Rewrite(Output);
```

注意：Text-oriented I/O 只在控制台程序中是可用的，也就是说，要在 Project Options 对话框的 Linker 页上选择“Generate console application”选项，或者在命令行编译器选项中使用-cc。在 GUI（非控制台）程序中，试图使用 Input 或 Output 进行读或写都将产生 I/O 错误。
一些工作于文本文件的 I/O 例程，不需要明确使用文件变量作为参数。若省略了文件参数，则依据过程或函数是输入还是输出，相应地使用默认地 Input 或 Output。比如，Read(X)对应于 Read(Input, X)，Write(X)对应于 Write(Output, X)。
调用一个作用于文本文件的输入或输出例程时，若你指定了一个文件，则必须使用 AssignFile 把它和一个外部文件相关联，并使用 Reset、Rewrite 或 Append 打开它。若你把一个用 Reset 打开的文件传给输出（写入）目的的过程或函数，则发生异常；反过来，若把用 Rewrite 或 Append 打开的文件传给输入（读取）目的的过程或函数，也将发生异常。

Untyped files（无类型文件）

无类型文件主要用于直接访问磁盘文件，而不考虑类型和结构，它是一种低级的 I/O 通道。无类型文件使用关键字 **file** 声明，没有其它内容。

```
var DataFile: file;
```

对无类型文件，Reset 和 Rewrite 过程允许你使用额外的参数来指定传输数据时记录的大小。因为历史原因，默认的记录大小是 128 字节。只有记录的大小为 1 时，才能正确反映任何文件的实际大小。
除了 Read 和 Write，所有作用于类型文件的过程和函数也能用于无类型文件。代替 Read 和 Write，BlockRead 和 BlockWrite 两个过程用于高速数据传输。

Text-file device drivers（文本文件设备驱动）

Text-file device drivers: Overview（概述）

你可以为程序定义自己的文本文件设备驱动。文本文件设备驱动由 4 个函数组成，它们完全实现了 Object Pascal 文件系统和一些设备间的接口。
这 4 个函数每一个定义一个设备驱动，它们是 *Open*、*InOut*、*Flush* 和 *Close*。每个函数的声明（函数头）都是

```
function DeviceFunc(var F: TTextRec): Integer;
```

这里，*DeviceFunc* 是函数的名称（也就是 *Open*、*InOut*、*Flush* 或 *Close*）。设备接口函数的返回值变成 IOResult 的返回值。若返回值是 0，则操作成功。
要把设备接口函数和指定的文件相关联，你必须写一个定制的 *Assign* 过程。*Assign* 过程必须把 4 个设备接口函数的地址赋给文本文件变量的的 4 个函数指针；并且，它必须在 Mode 字段存储 fmClosed“魔术”常量，在 BufSize 字段存储文本文件缓冲区的大小，在 BufPtr 字段存储指向文本文件缓冲区的指针，并且把 Name 字段清除。
比如，假设 4 个设备接口函数叫做 DevOpen、DevInOut、DevFlush 和 DevClose，*Assign* 过程应该看起来这样：

```
procedure AssignDev(var F: Text);
begin
  with TTextRec(F) do
  begin
    Mode := fmClosed;                //“魔术”常量
```

```
      BufSize := SizeOf(Buffer);         // 缓冲区大小
      BufPtr := @Buffer;                 // 缓冲区地址
      OpenFunc := @DevOpen;              // 把 4 个设备接口函数的地址赋给相应字段
      InOutFunc := @DevInOut;
      FlushFunc := @DevFlush;
      CloseFunc := @DevClose;
      Name[0] := #0;                     // 把 Name 清除
    end;
  end;
```

设备接口函数能使用文件记录（file record）中的 UserData 字段存储私有信息，在任何时候文件系统都不会修改这个字段。

Device functions（设备函数）

组成文本文件设备驱动的函数如下所述。

***Open*（打开）函数：**

Open 函数被 Reset、Rewrite 和 Append 标准过程调用，用来打开一个和设备关联的文本文件。在入口，Mode 字段包含 fmInput、fmOutput 或 fmInOut 来指示 Open 函数是否被 Reset、Rewrite 或 Append 调用。Open 函数根据 Mode 值来准备文件是被输入或输出。若指定了 fmInOut（表示 Open 函数被 Append 调用），在 Open 函数返回前，它必须被改为 fmOutput。

Open 总是在其它设备接口函数之前被调用，因为这个原因，*AssignDev* 只是初始化 OpenFunc 字段，把其余的工作（给字段赋函数地址）留给 *Open* 函数。基于 Mode 值，Open 能设置输入或输出目的的（函数）指针。通过判断当前的状态，可以省却 *InOut*、*Flush* 函数和 *CloseFile* 过程。

InOut（输入输出）函数：

在需要设备的输入输出时，InOut 函数会被 Read、Readln、Write、Writeln、Eof、Eoln、SeekEof、SeekEoln 和 CloseFile 标准例程调用。

当 Mode 为 fmInput 时，*InOut* 函数读取达到 BufSize 数目的字符到 BufPtr^中，然后返回读取的字符数目到 BufEnd；并且，它存储 0 到 BufPos 中。若 *InOut* 函数在响应输入请求时在 BufEnd 中返回 0，文件的 Eof 变为 True。

When Mode is fmOutput, the InOut function writes BufPos characters from BufPtr^, and returns zero in BufPos.

（当 Mode 为 fmOutput 时，InOut 函数从 BufPtr^写入 BufPos（BufSize？）个字符，并在 BufPos 中返回 0。）

Flush（清除缓冲区）函数：

每次在 Read、Readln、Write 和 Writeln 的最后调用 *Flush* 函数。它也可以清除文本文件缓冲区。

若 Mode 是 fmInput，*Flush* 函数能在 BufPos 和 BufEnd 存储 0，并清除缓冲区的剩余（未读取）字符，这个特征很少使用。

若 Mode 是 fmOutput，*Flush* 函数能像 InOut 函数一样把缓冲区的内容写入（设备），它保证写入设备的文本能立即出现在设备中。若 *Flush* 没做什么，直到缓冲区充满了或文件被关闭，文本才出现在设备中。

Close（关闭）函数

Close 函数被 CloseFile 标准过程调用，用来关闭和一个设备关联的文本文件。（若要打开的文件已经打开，则 Reset、Rewrite 和 Append 过程也会调用 *Close* 函数）若 Mode 是 fmOutput，在调用 *Close* 之前，文件

系统会调用 *InOut* 函数以保证所有字符被写入设备。

Handling null-terminated strings（处理零结尾字符串）

Object Pascal 扩展语法允许 Read、Readln、Str 和 Val 标准过程能应用于 0 基准字符数组，允许 Write、Writeln、Val、AssignFile 和 Rename 标准过程应用于 0 基准字符数组和字符指针。并且，提供下面的函数用来处理零结尾字符串。关于零结尾字符串的更多信息，请参考 Working with null-terminated strings。

函数	描述
StrAlloc	在堆中分配指定大小的字符缓冲区。
StrBufSize	返回用 StrAlloc 或 StrNew 分配的字符缓冲区的大小。
StrCat	连接两个字符串
StrComp	比较两个字符串
StrCopy	拷贝一个字符串
StrDispose	释放用 StrAlloc 或 StrNew 分配的字符缓冲区。
StrECopy	拷贝一个字符串并返回一个指向字符串末尾的指针。
StrEnd	返回一个指向字符串末尾的指针。
StrFmt	格式化一个或多个变量到一个字符串。
StrIComp	比较两个字符串，忽略大小写。
StrLCat	连接两个字符串，指定了目标（结果）字符串的最大长度。
StrLComp	比较两个字符串，指定了要比较的最大长度。
StrLCopy	拷贝字符串，指定了拷贝的最大长度。
StrLen	返回一个字符串的长度。
StrLFmt	格式化一个或多个变量到一个字符串，指定了字符串的最大长度。
StrLIComp	比较两个字符串，指定了要比较的最大长度，忽略大小写。
StrLower	转换字符串为小写形式。
StrMove	从一个字符串中移动一块字符到另一个字符串。
StrNew	在堆中分配一个字符串。
StrPCopy	拷贝一个 Pascal 字符串到一个零结尾字符串。
StrPLCopy	拷贝一个 Pascal 字符串到一个零结尾字符串，指定了字符串的最大长度。
StrPos	返回字符串中指定子串第一次出现的位置（指针）。
StrRScan	返回字符串中指定字符最后出现的位置（指针）。
StrScan	返回字符串中指定字符最先出现的位置（指针）。
StrUpper	转换字符串为大写形式。

标准字符串处理函数有针对于多字节的副本，它们也实现了对字符的 locale-specific 排序。多字节函数的名称以 Ansi-开始。比如，StrPos 的多字节版本是 AnsiStrPos。多字节字符的支持是和操作系统相关的，它基于本地设置（current locale）。

宽字符串

System 单元提供了三个函数，WideCharToString、WideCharLenToString 和 StringToWideChar，它们用来把 0 结束宽字符串转换为单字节或双字节长字符串。

关于宽字符串，参考 About extended character sets。

Other standard routines（其它标准例程）

下面的表格列出了经常使用的过程和函数，它们并不是完整的标准例程。

过程或函数	描述
Abort	Ends the process without reporting an error.
Addr	Returns a pointer to a specified object.
AllocMem	Allocates a memory block and initializes each byte to zero.
ArcTan	Calculates the arctangent of the given number.
Assert	Tests whether a boolean expression is True.
Assigned	Tests for a nil (unassigned) pointer or procedural variable.
Beep	Generates a standard beep using the computer speaker.
Break	Causes control to exit a for, while, or repeat statement.
ByteToCharIndex	Returns the position of the character containing a specified byte in a string.
Chr	Returns the character for a specified value.
Close	Terminates the association between a file variable and an external file.
CompareMem	Performs a binary comparison of two memory images.
CompareStr	Compares strings case sensitively.
CompareText	Compares strings by ordinal value and is not case sensitive.
Continue	Returns control to the next iteration of for, while, or repeat statements.
Copy	Returns a substring of a string or a segment of a dynamic array.
Cos	Calculates the cosine of an angle.
CurrToStr	Converts a currency variable to a string.
Date	Returns the current date.
DateTimeToStr	Converts a variable of type TDateTime to a string.
DateToStr	Converts a variable of type TDateTime to a string.
Dec	Decrements an ordinal variable.
Dispose	Releases memory allocated for a dynamic variable.
ExceptAddr	Returns the address at which the current exception was raised.
Exit	Exits from the current procedure.
Exp	Calculates the exponential of X.
FillChar	Fills contiguous bytes with a specified value.
Finalize	Uninitializes a dynamically allocated variable.
FloatToStr	Converts a floating point value to a string.
FloatToStrF	Converts a floating point value to a string, using specified format.
FmtLoadStr	Returns formatted output using a resourced format string.
FmtStr	Assembles a formatted string from a series of arrays.
Format	Assembles a string from a format string and a series of arrays.
FormatDateTime	Formats a date-and-time value.
FormatFloat	Formats a floating point value.
FreeMem	Disposes of a dynamic variable.
GetMem	Creates a dynamic variable and a pointer to the address of the block.
GetParentForm	Returns the form or property page that contains a specified control.
Halt	Initiates abnormal termination of a program.
Hi	Returns the high-order byte of an expression as an unsigned value.

High	Returns the highest value in the range of a type, array, or string.
Inc	Increments an ordinal variable.
Initialize	Initializes a dynamically allocated variable.
Insert	Inserts a substring at a specified point in a string.
Int	Returns the integer part of a real number.
IntToStr	Converts an integer to a string.
Length	Returns the length of a string or array.
Lo	Returns the low-order byte of an expression as an unsigned value.
Low	Returns the lowest value in the range of a type, array, or string.
LowerCase	Converts an ASCII string to lowercase.
MaxIntValue	Returns the largest signed value in an integer array.
MaxValue	Returns the largest signed value in an array.
MinIntValue	Returns the smallest signed value in an integer array.
MinValue	Returns smallest signed value in an array.
New	Creates a new dynamic variable and references it with a specified pointer.
Now	Returns the current date and time.
Ord	Returns the ordinal value of an ordinal-type expression.
Pos	Returns the index of the first character of a specified substring in a string.
Pred	Returns the predecessor of an ordinal value.
Ptr	Converts a specified address to a pointer.
Random	Generates random numbers within a specified range.
ReallocMem	Reallocates a dynamic variable.
Round	Returns the value of a real rounded to the nearest whole number.
SetLength	Sets the dynamic length of a string variable or array.
SetString	Sets the contents and length of the given string.
ShowException	Displays an exception message with its address.
ShowMessage	Displays a message box with an unformatted string and an OK button.
ShowMessageFmt	Displays a message box with a formatted string and an OK button.
Sin	Returns the sine of an angle in radians.
SizeOf	Returns the number of bytes occupied by a variable or type.
Sqr	Returns the square of a number.
Sqrt	Returns the square root of a number.
Str	Formats a string and returns it to a variable.
StrToCurr	Converts a string to a currency value.
StrToDate	Converts a string to a date format (TDateTime).
StrToDateTime	Converts a string to a TDateTime.
StrToFloat	Converts a string to a floating-point value.
StrToInt	Converts a string to an integer.
StrToTime	Converts a string to a time format (TDateTime).
StrUpper	Returns a string in upper case.
Succ	Returns the successor of an ordinal value.
Sum	Returns the sum of the elements from an array.
Time	Returns the current time.
TimeToStr	Converts a variable of type TDateTime to a string.
Trunc	Truncates a real number to an integer.

UniqueString	Ensures that a string has only one reference. (The string may be copied to produce a single reference.)
UpCase	Converts a character to uppercase.
UpperCase	Returns a string in uppercase.
VarArrayCreate	Creates a Variant array.
VarArrayDimCount	Returns number of dimensions of a Variant array.
VarARrayHighBound	Returns high bound for a dimension in a Variant array.
VarArrayLock	Locks a Variant array and returns a pointer to the data.
VarArrayLowBound	Returns the low bound of a dimension in a Variant array.
VarArrayOf	Creates and fills a one-dimensional Variant array.
VarArrayRedim	Resizes a Variant array.
VarArrayRef	Returns a reference to the passed Variant array.
VarArrayUnlock	Unlocks a Variant array.
VarAsType	Converts a Variant to specified type.
VarCast	Converts a Variant to a specified type, storing the result in a variable.
VarClear	Clears a Variant.
VarCopy	Copies a Variant.
VarToStr	Converts Variant to string.
VarType	Returns type code of specified Variant.

关于格式化字符串的信息，请参考 Format strings。

Libraries and packages（库和包）

Libraries and packages: Overview（概述）

动态调入库（dynamically loadable library）在 Windows 下是一个动态链接库（dynamic-link library，DLL），在 Linux 下是一个共享目标库（shared object library）。它是一个例程集合，程序以及其它动态链接库和共享目标库能够调用这些例程。像单元一样，动态调入库包含共享的代码或资源，但这种库是一个单独编译的可执行文件，它在运行时被链接到使用它的程序中。
为了区分它们和独立的可执行文件，在 Windows 下包含编译 DLL 的文件扩展名是.DLL；在 Linux 下包含共享目标的文件扩展名是.so。Object Pascal 程序可以调用由其它语言编写的 DLL 或共享目标，其它语言编写的程序也可以调用由 Object Pascal 编写的 DLL 或共享目标。
（这里是翻译为对象，还是目标呢？）

Calling dynamically loadable libraries（调用动态调入库）

Calling dynamically loadable libraries（调用动态调入库）

你可以直接调用操作系统的例程，但它们直到运行时才被链接到你的程序。这说明在编译程序时它们不必存在，同时也说明在（试图）导入一个例程时不会进行编译时验证。
在调用由共享目标文件（shared object，是仅仅指 Linux 下的吗？）所定义的例程前，你必须导入这些例程。这有两种方式：一是声明一个外部（**external**）过程或函数，二是直接调用操作系统。不论使用哪种方式，例程都是直到运行时才链接到你的程序中。
Object Pascal 不支持从共享库中导入变量。

静态调入
导入过程或函数最简单的方法是用 **external** 指示字声明它们，比如，

在 Windows 下： **procedure** DoSomething; **external** 'MYLIB.DLL';
在 Linux 下： **procedure** DoSomething; **external** 'mylib.so';

若你在程序中包含这个声明，MYLIB.DLL（Windows）或 mylib.so（Linux）在程序启动时被调入一次，在程序的整个运行期间，标志符 DoSomething 总是指同一个共享库中的同一个入口点。
导入例程的声明可以直接放在需要它们的程序或单元中。但为了维护方便，你可以把 **external** 声明放在一个单独的“导入单元”中，这个单元也可以包含和库进行交互所需要的常量和类型声明。其它使用这个导入单元的模块就可以调用它声明的任何例程。
要了解 **external** 声明的信息，请参考 External declarations。

Dynamic loading（动态调入）

你可以直接调用操作系统的库函数来访问一个库中的例程，这些库函数包括 LoadLibrary、FreeLibrary 和 GetProcAddress。在 Windows 下，这些函数在 Windows.pas 单元声明，在 Linux 下，为了兼容性考虑，它们在 SysUtils.pas 单元实现，实际的 Linux 例程是 dlopen、dlclose 和 dlsym（这些都在 Kylix 的 Libc 单元声明，请参考帮助）。此时，我们使用过程类型的变量来引用导入的例程。
比如，在 Windows 或 Linux 下：

```
uses Windows, ...; {On Linux, replace Windows with SysUtils }
type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
  TGetTime = procedure(var Time: TTimeRec);
  THandle = Integer;
var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ...
begin
  Handle := LoadLibrary('libraryname');
  if Handle <> 0 then
  begin
    @GetTime := GetProcAddress(Handle, 'GetTime');
    if @GetTime <> nil then
    begin
      GetTime(Time);
      with Time do
        WriteLn('The time is ', Hour, ':', Minute, ':', Second);
    end;
    FreeLibrary(Handle);
  end;
end;
```

当你用这种方式导入例程时，直到 LoadLibrary 调用开始执行，库才被调入，库后来通过调用 FreeLibrary 进行释放。这使你能节省内存，并且在某些需要的库不存在的情况下也能运行你的程序。

同样的例子在 Linux 下也可以这样实现：

```
uses Libc, ...;
type
  TTimeRec = record
    Second: Integer;
    Minute: Integer;
    Hour: Integer;
  end;
  TGetTime = procedure(var Time: TTimeRec);
  THandle = Pointer;
var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;
  ...
begin
```

```
    Handle := dlopen('datetime.so', RTLD_LAZY);
    if Handle <> 0 then
    begin
      @GetTime := dlsym(Handle, 'GetTime');
      if @GetTime <> nil then
      begin
        GetTime(Time);
        with Time do
          WriteLn('The time is ', Hour, ':', Minute, ':', Second);
      end;
      dlclose(Handle);
    end;
  end;
```

采用这种方式导入例程，直到 dlopen 调用开始执行，共享目标文件才被调入，目标文件后来通过调用 dlclose 进行释放。这使你能节省内存，并且在某些需要的库不存在的情况下也能运行你的程序。

Writing dynamically loadable libraries（编写动态调入库）

Writing dynamically loadable libraries（编写动态调入库）

动态调入库的主源文件和程序的一样，除了它以关键字 **library** 开始（取代 **program**）。
只有被库明确输出的例程才能被其它库或程序导入，下面的例子演示了库输出两个函数，Min 和 Max。

```
  library MinMax;
  function Min(X, Y: Integer): Integer; stdcall;
  begin
    if X < Y then Min := X else Min := Y;
  end;
  function Max(X, Y: Integer): Integer; stdcall;
  begin
    if X > Y then Max := X else Max := Y;
  end;
  exports
    Min,
    Max;
  begin
  end.
```

若要你的库对其它语言编写的程序是可见的，最安全的办法是在声明输出函数时指定 **stdcall** 调用约定，其它语言或许不支持 Object Pascal 默认的 **register** 调用约定。

Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a uses clause, an exports clause, and the initialization code. For example,

库可以通过多个单元文件创建，此时，库的源文件通常简化为包含一个 **uses** 子句、一个 **exports** 子句和初始化代码。比如，

```
  library Editors;
  uses EdInit, EdInOut, EdFormat, EdPrint;
  exports
```

```
    InitEditors,
    DoneEditors name Done,
    InsertText name Insert,
    DeleteSelection name Delete,
    FormatSelection,
    PrintSelection name Print,
     ...
    SetErrorHandler;
  begin
    InitLibrary;
  end.
```

你可以把 **exports** 子句放在单元的接口或实现部分，任何在 **uses** 子句中包含这个单元的库自动输出单元输出的例程，不必有自己的 **exports** 子句。

指示字 **local** 标记一个例程不能被输出，它是平台相关的，在 Windows 编程中没有作用。

在 Linux 下，**local** 指示字对编译进库但却不输出的例程提供了一点点性能优化。这个指示字用于标准的过程和函数，不能用于方法。比如，一个用 **local** 声明的例程，

```
  function Contraband(I: Integer): Integer; local;
```

它不刷新 EBX 寄存器，因此

- 它不能从一个库输出；
- 它不能在单元的接口部分声明；
- 不能取得它的地址或赋给一个过程类型的变量；
- 若它完全是一个汇编语言例程，除非调用者设置了 EBX 寄存器，否则不能从其它单元调用它

The exports clause（exports 子句）

当一个例程在 exports 子句中列出时，它将被输出，它的格式如下

```
  exports entry1, ..., entryn;
```

这里，每个 entry 包括一个过程、函数或变量（它必须在 **exports** 子句之前声明）的名称，后面跟参数列表（只有当输出重载的例程时）和一个可选的 **name** 说明符，你可以使用单元名限定过程或函数的名称。（入口也可以包含指示字 resident，它是为了向后兼容性，编译器将忽略它）

只有在 Windows 下能使用索引说明符，它包括指示字 **index**，后面跟一个介于 1 到 2,147,483,647 之间的数字常量（为提高程序效率，使用较小的索引值）。若入口中没有指定索引，在输出表中例程被自动赋予一个号码。

注意：索引说明符只是为了向后兼容性，不鼓励使用，在其它开发工具中可能引起问题。

名称说明符包括指示字 **name**，后面跟一个字符串常量。若入口没有名称说明符，例程被输出时使用声明的原始名称，包括拼写和大小写。当要使用不同的名称输出一个例程时，使用 name 子句。比如，

```
  exports
    DoSomethingABC name 'DoSomething';
```

当在动态调入库中输出重载的函数或过程时，你必须在 **exports** 子句中指定它的参数列表，比如，

```
  exports
    Divide(X, Y: Integer) name 'Divide_Ints',
    Divide(X, Y: Real) name 'Divide_Reals';
```

在 Windows 下，不要在重载的例程入口中使用 **index** 说明符。

exports 子句可出现在程序或库声明部分的任何位置，次数也不受限制，同样，当出现在单元的接口或实现部分时，情况也是如此。程序很少包含 **exports** 子句。

Library initialization code（库初始化代码）

一个库的块（block）所包含的语句构成了库的初始化代码，每当库被调入时，这些代码执行一次。它们的典型任务包括注册窗口类和初始化变量等。库的初始化代码也可以使用 **ExitProc** 变量安装一个退出过程（exit procedure），就像在 Exit procedures 中描述的那样。退出过程在库被卸载时执行。
库的初始化代码通过设定 **ExitCode** 变量为非 0 来标记一个错误。**ExitCode** 在 System 单元声明，默认值时 0。若库的初始化代码把 ExitCode 设置为其它值，库将被卸载，调用程序被通知发生了错误。类似地，若初始化代码执行中发生了未处理的异常，调用程序也将被通知调入库时失败。
这里是一个关于库初始化代码和退出过程的例子。

```
library Test;
var
  SaveExit: Pointer;
procedure LibExit;
begin
  ...                           // 库的退出代码
  ExitProc := SaveExit;         // 恢复退出过程链表
end;
begin
  ...                           // 库的初始化代码
  SaveExit := ExitProc;         // 保存退出过程链表
  ExitProc := @LibExit;         // 设置 LibExit 为退出过程
end.
```

当库被卸载时，通过重复调用存储在 **ExitProc** 中的地址执行退出过程，直到 **ExitProc** 变成 **nil**。所有被库使用的单元，它们的初始化代码在库的初始化代码执行之前被执行，它们的结束化部分在库的退出过程执行后才执行。

Global variables in a library（库中的全局变量）

在共享库中声明的全局变量不能被 Object Pascal 程序导入。
一个库一次能被多个程序使用，但每个程序在自己的进程空间中有一个库拷贝，且每个拷贝有自己的全局变量集合。对于在多个库间（或一个库的多个实例间）共享内存，它们必须使用内存映射文件。更进一步的信息，请参考系统文档。

Libraries and system variables（库和系统变量）

在 System 单元声明的几个变量对那些程序库有特殊影响。使用 **IsLibrary** 变量来确定代码是作为程序还是库执行，**IsLibrary** 在程序中总是 **True**，在库中总是 **False**。在库的生命期内，**HInstance** 存储了它的实例句柄，**CmdLine** 在库中总是 **nil**。
DLLProc 变量允许一个库监测操作系统对它的入口点（entry point）的调用，这个特征通常只是由支持多线程的库使用。DLLProc 在 Windows 和 Linux 下都存在，但用起来不同。在 Windows 下，DLLProc 用于多线程程序，在 Linux 下，它用来判断库何时被卸载。对所有的退出行为，你应该使用 finalization sections，而不是退出过程。
要监测操作系统调用，创建一个回调过程，它接收一个整数参数，比如，

```
procedure DLLHandler(Reason: Integer);
```

然后把过程地址赋给 DLLProc 变量。当过程被调用时，它（参数）被赋予如下值：

DLL_PROCESS_DETACH	表明库从调用进程的地址空间分离，这是一个 clean exit 或调用 FreeLibrary（在 Linux 下是 dlclose）的结果。
DLL_THREAD_ATTACH	表明当前进程正创建一个新线程（Windows）
DLL_THREAD_DETACH	表明一个线程（干净地）结束（Windows）

在 Linux 下，这些在 Libc 单元定义。
在过程体中，你能依据哪个参数被传递给过程来指定要采取地行动。

Exceptions and runtime errors in libraries（库的异常和运行时错误）

当在动态调入库中有异常发生但没处理时，它传播到库的外面到达调用者。如果调用程序（或库）本身是用 Object Pascal 编写的，可通过标准的 **try...except** 语句处理它。
注意：在 Linux 下，只有当库和程序使用同一套运行包（包含 EH 代码）创建或者都链接到 ShareExcept 时才可以。
若调用程序（或库）是用其它语言编写的，异常被当作操作系统的异常（异常代码：$0EEDFACE）进行处理。在操作系统异常记录的 ExceptionInformation 数组的第一个入口中，包含了异常地址，第二个入口包含一个指向 Object Pascal 异常对象的引用。
通常，你不应该使异常扩散到库的外面。在 Windows 下，Delphi 异常映射到操作系统的异常模型，Linux 没有异常模型。
若一个库没有使用 SysUtils 单元，它不支持异常处理。这种情况下，若库发生运行时错误，调用程序将终止。因为库没有办法知道它是否从一个 Object Pascal 程序进行调用，它不能调用程序的退出过程，程序只是简单地被终止，并从内存中清除。

Shared-memory manager (共享内存管理器)

在 Windows 下，若 DLL 输出的例程以长字符串或动态数组作为参数或者作为函数的返回值（不管是直接的，还是通过记录或对象封装的），那么，DLL 和它的客户程序（或 DLL）必须使用 ShareMem 单元；当一个程序或 DLL 调用 New 或 GetMem 分配内存，而在另一个模块中调用 Dispose 或 FreeMem 来释放内存时，上面的规则同样适用。ShareMem 单元应当在程序或库的 **uses** 子句中第一个列出。
ShareMem 是 BORLANDMM.DLL 内存管理器的接口单元，它允许在模块间共享动态分配的内存。BORLANDMM.DLL 必须连同使用 ShareMem 单元的程序和 DLL 一同发布。当程序或 DLL 使用 ShareMem 时，它的内存管理器被 BORLANDMM.DLL 中的取代。
Linux 使用 glibc 的 malloc 来管理共享内存。

Packages（包）

Packages: Overview（概述）

包是一个特殊编译库，它被用于程序、IDE 或者两者同时使用。包允许你在不影响源代码的情况下重新安排代码存在的时机，这有时称为 application partitioning。（Packages allow you to rearrange when code resides without affecting the source code. This is sometimes referred to as application partitioning.）
运行时包在程序运行时提供必要功能，设计时包用来在 IDE 中安装组件，并且为定制的组件创建特殊的属性编辑器。一个包能同时作用于设计时和运行时，设计时包经常要在它们的 **requires** 子句中引用运行

时包才能工作。

为了区分包和其它库，包被存储在文件

- 在 Windows 下，包的扩展名是.bpl（Borland package library）
- 在 Linux 下，包通常以前缀 bpl 开始，扩展名是.so。

通常，程序启动时包被静态调入，但你可以使用 LoadPackage 和 UnloadPackage 例程（在 SysUtils 单元）来动态调入包。

注意：当程序利用包时，被打包的每个单元（名）还是要出现在使用它的每个源文件的 **uses** 子句中。

Package declarations and source files（包声明和源文件）

Package declarations and source files（包声明和源文件）

每个包用一个单独的源文件进行声明，它的扩展名是.dpk，以便和其它包含 Object Pascal 代码的文件混淆。包源文件不包括类型、数据、过程或函数声明。取而代之的是，它包含

- 包的名称；
- 它所需要的其它包的列表。这些包被链接到新包中；
- 包被编译时所包含的（或绑定的）单元文件列表。包实际上是这些代码单元的一个外包装，这些单元为编译后的包提供功能。

包的声明有如下形式：

```
package packageName;
  requiresClause;
  containsClause;
end.
```

这里，packageName 是任何有效标志符；*requiresClause* 和 *containsClause* 都是可选的。比如，下面的代码声明了 DATAX 包。

```
package DATAX;
requires
  baseclx,
  visualclx;
  contains Db, DBLocal, DBXpress, ... ;
end.
```

requires 子句列出了声明的包所需要的其它外部包。它包括指示字 **requires**，后面是逗号隔开的包名称，然后跟一个分号。若包不引用其它包，它不需要 **requires** 子句。

contains 子句指明要被编译并绑定到包中的单元。它包括指示字 contains，后面是逗号隔开的单元名列表，然后跟一个分号。单元名后面可以跟关键字 **in** 和源文件名，源文件名可以包括或不包括路径，并用单引号括起来，路径可以是绝对的，也可以是相对路径。比如，

```
contains MyUnit in 'C:\MyProject\MyUnit.pas';                // Windows
contains MyUnit in '\home\developer\MyProject\MyUnit.pas';    // Linux
```

注意：包单元中的线程局部变量（用 **threadvar** 声明）不能被使用包的客户访问。

Naming packages（命名包）

一个编译包包括几个生成文件。比如，包 DATAX 的源文件是 DATAX.dpk，编译器从它生成一个可执行的二进制映像文件，它叫做

- 在 Windows 下： DATAX.bpl and DATAX.dcp
- 在 Linux 下： bplDATAX.so and DATAX.dcp.

DATAX 用来在其它包的 **requires** 子句中指定这个包，或当程序使用这个包时用 DATAX 来指定。在一个工程中，包名必须是唯一的。

The requires clause（requires 子句）

requires 子句列出了当前包所使用的其它外部包，它的作用就像单元文件中的 **uses** 子句。当程序使用当前包，并且使用了 **requires** 子句中列出的外部包所包含的一个单元时，外部包被自动链接到程序。
若包中的单元文件引用了其它包中的单元，则其它包应该被包含在第一个包的 **requires** 子句中。若在 **requires** 子句中省略了其它包，编译器从.dcu（Windows）或.dpu（Linux）文件调入引用单元。

避免循环包引用
包在它们的 **requires** 子句中不能包含循环引用。它的意思是

- 包在它的 **requires** 子句中不能引用自己；
- 引用链结束时不能引用链中的任何包。若包 A 需要包 B，那么 B 不能需要 A；若 A 需要 B，而 B 需要 C，那么 C 不能需要 A。

重复包引用
在包的 **requires** 子句中，编译器忽略重复的包引用，但为了设计及程序的可读性考虑，应当删除重复的包引用。

The contains clause（contains 子句）

contains 子句指明了绑定到包中的单元。不要在 **contains** 子句中包含文件扩展名。

避免使用多余的源代码
包不能出现在其它包的 **contains** 子句中，也不能出现在单元的 **uses** 子句中。
直接在包的 **contains** 子句中所包含的单元，以及（间接的）这些单元所使用的单元，在编译时都被绑定到包中。一个包所包含的任何单元（直接的或间接的），不能被它的 **requires** 子句中所需要的包再包含。
一个单元不能被包含（直接或间接）在同一个程序所需要的多个包中。

Compiling packages（编译包）

Compiling packages: Overview（概述）

我们通常使用包编辑器创建.dpk 文件，然后从 IDE 编译一个包，也可以使用命令行直接编译一个.dpk 文件。当创建的工程包含一个包时，若需要的话，包会自动（暗中）被重新编译。

Generated files（生成文件）

下面的表格列出了成功编译一个包时所产生的文件

文件扩展名	内容
dcp	一个二进制映像文件，它包含一个package header和包中所有的dcu（Windows）或 dpu（Linux）构成的 concatenation。每个包文件创建一个 dcp 文件，它的文件名和 dpk 源文件名相同。
dcu (Windows) dpu (Linux)	和包的一个单元对应的二进制映像文件。若需要的话，为每个单元文件创建一个 dcu 或 dpu 文件。
.bpl on Windows bpl<package>.so on Linux	运行时包。它是一个特殊的共享库。它的文件名和 dpk 源文件名相同。

Package-spcific compiler directives（库编译器指示字）

下面的表格，列出了能被插入到源代码中的适用于包的编译器指示字。

指示字	作用
{$IMPLICITBUILD OFF}	防止一个包在以后被重新编译。用于提供低级功能、不会经常变化，或源代码不会被发布的.dpk 文件。
{$G–} or {$IMPORTEDDATA OFF}	Disables creation of imported data references. This directive increases memory-access efficiency, but prevents the unit where it occurs from referencing variables in other packages.
{$WEAKPACKAGEUNIT ON}	Packages unit weakly.
{$DENYPACKAGEUNIT ON}	Prevents unit from being placed in a package.
{$DESIGNONLY ON}	Compiles the package for installation in the IDE. (Put in .dpk file.)
{$RUNONLY ON}	Compiles the package as runtime only. (Put in .dpk file.)

在源文件中包含**{$DENYPACKAGEUNIT ON}**，能防止单元文件被编译到包中。包含**{$G–}**或**{$IMPORTEDDATA OFF}**能防止一个包和其它包用在同一个程序中。
如果合适，其它编译器指示字也可以包含在包源代码中。

Package-spcific command-line compiler switches（库命令行编译开关）

下面是适合于包的命令行编译开关。

开关	作用
–$G–	Disables creation of imported data references. Using this switch increases memory-access efficiency, but prevents packages compiled with it from referencing variables in other packages.
–LE path	Specifies the directory where the compiled package file will be placed.
–LN path	Specifies the directory where the package dcp file will be placed
–LU packageName [;packageName2;…]	Specifies additional runtime packages to use in an application. Used when compiling a project.
–Z	Prevents a package from being implicitly recompiled later. Use when compiling packages that provide low-level functionality, that change infrequently between builds, or whose source code will not be distributed.

使用–G–开关能防止一个包和其它包用于同一个程序中。
如果合适，编译包时其它命令行选项也可以使用。

Object interfaces（对象接口）

Object interfaces: Overview（概述）

对象接口（或简单地说接口）定义了能被一个类实现的方法。接口声明和类相似，但不能直接实例化它，也不能自己实现（定义）它们的方法，而是由支持接口的任何类来提供实现。一个接口类型的变量能引用一个实现了此接口的对象，但是，只有接口中声明的方法才能通过这个变量进行调用。
接口提供了一些多继承的好处，却没有多继承带来的语义困难。它们对使用分布式对象模型（）也是非常有效的，定制的、支持接口的对象可以和其它语言（比如 C++、Java 和其它语言）编写的对象进行交互。

Interface types（接口类型）

Interface types: Overview（概述）

像类一样，接口只能在程序或单元的最外层声明，而不能在过程或函数中声明。一个接口类型的声明有如下格式

```
type interfaceName = interface (ancestorInterface)
  ['{GUID}']
  memberList
end;
```

这里，(*ancestorInterface*)和['{GUID}']是可选的。在大多数方面，接口声明和类声明类似，但有以下限制：

- *memberList* 只包括方法和属性，字段在接口中是不允许的；
- 因为接口没有字段，所以属性的读（**read**）和写（**write**）限定符必须是方法；
- 接口的所有成员都是公有的（public），不允许使用可见性限定符和存储限定符（但一个数组属性能被声明为 **default**）；
- 接口没有构造函数和析构函数，它们不能被（直接）实例化，除非使用实现了它们（的方法）的类；
- 方法不能被声明为 **virtual**、**dynamic**、**abstract** 或 **override**。因为接口自己不实现它们的方法，这些声明没有意义。

这里是个接口声明的例子：

```
type
  IMalloc = interface(IInterface)
    ['{00000002-0000-0000-C000-000000000046}']
    function Alloc(Size: Integer): Pointer; stdcall;
    function Realloc(P: Pointer; Size: Integer): Pointer; stdcall;
    procedure Free(P: Pointer); stdcall;
    function GetSize(P: Pointer): Integer; stdcall;
    function DidAlloc(P: Pointer): Integer; stdcall;
    procedure HeapMinimize; stdcall;
  end;
```

在某些接口声明中，**interface** 关键字被换成了 **dispinterface**，这种构造（连同 **dispid**、**readonly** 和 **writeonly** 指示字）是平台相关的，不能在 Linux 程序中使用。

IInterface and inheritance（接口和继承）

接口和类一样，继承它的祖先所有的方法；但接口不象类，它们不实现方法。一个接口继承的是实现这些方法的义务，把这个义务委托给支持此接口的任何一个类。
声明一个接口时可以指定一个祖先接口，如果没有指明的话，则它直接继承自 IInterface。IInterface 在 System 单元定义，是其它所有接口的根类。IInterface 定义了三个方法：QueryInterface、_AddRef 和 _Release。
注意：IInterface 和 IUnknown 是相同的。考虑到平台无关性，通常要使用 IInterface；IUnknown 最好用在一些特殊的程序中，它依赖于 Windows 平台。
QueryInterface 支持在一个对象所实现的不同接口之间自由跳转；_AddRef 和_Release 为接口引用提供生命期内存管理。实现这三个方法最简单的方式是从 TInterfacedObject（在 System 单元声明）派生一个类。若在实现这些方法时使用空函数，就可以忽略它们，但 COM 对象（只适用于 Windows）必须通过_AddRef 和_Release 进行管理。

Interface identification（接口标志）

一个接口声明可指定一个全局唯一标识符（GUID），它用一个被中括号括起来的字符串表示，它出现在接口的成员之前。GUID 部分以如下形式声明：
['{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}']
这里，每个 x 是一个十六进制的位（0 到 9 或者 A 到 F）。在 Windows 中，类型库编辑器（Type Library editor）能为新接口自动产生 GUID，你也可以在代码编辑器（Code editor）中使用 Ctrl+Shift+G 来创建 GUID（在 Linux 中，你必须使用 Ctrl+Shift+G）。
GUID 是一个 16 字节的二进制数，它唯一地标识一个接口。如果一个接口有 GUID，则可以通过查询接口来获得它的实现的引用。
TGUID 和 PGUID 在 System 单元声明，用来对 GUID 进行操作。

```
type
  PGUID = ^TGUID;
  TGUID = packed record
    D1: Longword;
    D2: Word;
    D3: Word;
    D4: array[0..7] of Byte;
  end;
```

当你声明一个 TGUID 类型的常量时，可以用字符表示它的值，例如

```
const IID_IMalloc: TGUID = '{00000002-0000-0000-C000-000000000046}';
```

在调用过程或函数时，GUID 或者一个接口的名称（标识）都可以作为 TGUID 类型的值参或常量参数。例如下面的声明

```
function Supports(Unknown: IInterface; const IID: TGUID): Boolean;
```

Supports 函数可以用下面的两种方法进行调用：

```
if Supports(Allocator, IMalloc) then ...        //接口名称
if Supports(Allocator, IID_IMalloc) then ...    //GUID 常量
```

Calling conventions for interfaces（接口调用约定）

默认的调用约定是 **register**，但当接口在程序模块（尤其当它们用其它语言编写时）间共享时，需要声明所有的方法为 **stdcall** 调用方式；实现 CORBA 接口时使用 safecall 调用约定；在 Windows 下，你可以用 safecall 来实现双重调度接口的方法。
关于调用约定的更多信息，请参考 Calling conventions。

Interface properties（接口属性）

接口声明的属性只能通过接口类型的表达式进行访问，类类型的变量不行；并且，接口的属性只在接口被编译的程序中是可见的。比如，在 Windows 下，COM 对象没有属性。
在接口中，属性的读和写必须通过方法来完成，因为不存在字段。

Forward declarations（Forward 声明）

若声明一个接口时以 **interface** 关键字和一个分号结束，没有指定它的祖先、GUID 以及成员列表，这是一个 forward 声明。Forward 声明的接口必须在同一个声明区域进行定义声明，换句话说，在 forward 声明和它的定义声明之间除了类型声明外，不能有任何其它内容。
Forward 声明允许定义相互依赖的接口，例如

```
type
  IControl = interface;
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000044}']
    function GetControl(Index: Integer): IControl;
    ...
  end;
  IControl = interface
    ['{00000115-0000-0000-C000-000000000049}']
    function GetWindow: IWindow;
    ...
  end;
```

相互继承（派生）的接口是不允许的。比如，从 IControl 派生 IWindow，又从 IWindow 派生 IControl 是非法的。

Implementing interfaces（实现接口）

Implementing interfaces（实现接口）

一旦声明一个接口，在使用之前必须通过一个类来实现它。实现接口的类必须在声明时指定接口，它出现在父类的名称之后。声明有如下格式

```
type className = class (ancestorClass, interface1, ..., interfacen)
  memberList
 end;
```

比如，

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    ...
  end;
```

声明了一个叫做 TMemoryManager 的类，它实现了 IMalloc 和 IErrorInfo 接口。当一个类实现某个接口时，它必须实现（或通过继承实现）接口声明的每个方法。

下面是 System 单元中 TInterfacedObject 的声明：

```
type
TInterfacedObject = class(TObject, IInterface)
protected
    FRefCount: Integer;
    function QueryInterface(const IID: TGUID; out Obj): HResult; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
public
    procedure AfterConstruction; override;
    procedure BeforeDestruction; override;
    class function NewInstance: TObject; override;
    property RefCount: Integer read FRefCount;
end;
```

TInterfacedObject 实现了 IInterface 接口，因此，TInterfacedObject 声明和实现了 IInterface 的三个方法。

实现接口的类可以被当作基类（前面第一个例子中，TmemoryManager 就是以 TInterfacedObject 作为基类）。因为每个接口都继承自 IInterface，所以，一个实现接口的类必须实现 QueryInterface、_AddRef 和 _Release 方法。System 单元中的 TInterfacedObject 实现了这些方法，所以，其它实现接口的类可以方便地通过继承它来实现。

当一个接口被实现时，它声明的每个方法都将和它的实现类中的方法一一对应：有相同的返回值类型、相同的调用约定、同样数目的参数，并且参数的类型和位置是相同的。默认情况下，接口中的方法对应于实现类中同名的方法。

Method resolution clauses（方法解析子句，方法别名）

在声明类时，通过方法解析子句你可以改变默认的名称映射。当一个类实现两个或多个接口、并且其中有同名的方法时，使用方法别名可以避免混乱。

方法解析子句采用如下格式

```
procedure interface.interfaceMethod = implementingMethod;
```

或

```
function interface.interfaceMethod = implementingMethod;
```

这里，*implementingMethod* 是在这个类或它的一个祖先类中声明的方法。*implementingMethod* 可以是此类中稍后声明的方法；当 *implementingMethod* 是祖先类的方法时，若祖先类不在同一个模块中，则方法不能是私有的。（因为不能访问另一个模块中的私有成员）

比如，下面的类声明

```
type
  TMemoryManager = class(TInterfacedObject, IMalloc, IErrorInfo)
    function IMalloc.Alloc = Allocate;
    procedure IMalloc.Free = Deallocate;
```

```
    …
  end;
```

把 IMalloc 接口的 Alloc 和 Free 方法分别映射到 TMemoryManager 类的 Allocate 和 Deallocate 方法。
方法解析子句不能更改祖先类引入的方法映射。

Changing inherited implemetations（更改继承实现）

通过覆盖实现方法，子类可以改变接口方法的实现方式，这要求实现方法是虚拟的或者是动态的。
一个类也可以重新（完整地）实现它从祖先类继承下来的接口，这要求它在声明时重新列出这个接口。
例如

```
type
  IWindow = interface
    ['{00000115-0000-0000-C000-000000000146}']
    procedure Draw;
    …
  end;
  TWindow = class(TInterfacedObject, IWindow)      // Twindow 实现 IWindow
    procedure Draw;
    …
  end;
  TFrameWindow = class(TWindow, IWindow)           //  TframeWindow  重 新 实 现
IWindow
    procedure Draw;                                // 这是静态方法，隐藏了父类的方法
    …
  end;
```

重新实现一个接口隐藏了它继承下来的实现，因此，祖先类中的方法解析子句对于重新实现的接口没有影响。

Implementing interfaces by delegation（通过委托实现接口）

Implementing interfaces by delegation（通过委托实现接口）

implements 指示字允许你在实现类中委托一个属性来实现接口，比如

```
property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
```

上面声明了一个叫做 MyInterface 的属性，它实现了接口 IMyInterface。
在属性声明中，**implements** 指示字必须是最后一项，它可以实现多个接口，接口之间以逗号分隔。委托的属性要满足以下条件：

- 必须是类或接口类型；
- 不能是数组属性，也不能使用 **index** 限定符；
- 必须有一个读限定符。若属性使用 **read** 方法，则方法必须使用默认的 **register** 调用约定，并且不能是动态方法（但可以是虚方法），也不能使用 **message** 指示字。

注意：实现委托接口的类应当从 TAggregatedObject 派生。

Delegating to an interface-type property（委托一个接口类型的属性）

如果委托的属性是接口类型，那么此接口（或者它的派生接口）必须出现在类声明中的祖先列表中（也就是声明实现这些接口）。委托的属性必须返回一个对象，此对象所属的类完全实现了 **implements** 所指明的接口，并且没有使用方法解析子句。比如

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
  TMyClass = class(TObject, IMyInterface)
    FMyInterface: IMyInterface;
    property MyInterface: IMyInterface read FMyInterface implements IMyInterface;
  end;
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyInterface := ...     // 某个实现了 IMyInterface 接口的对象
  MyInterface := MyClass;
  MyInterface.P1;
end;
```

Delegating to a class-type property（委托一个类类型属性）

如果委托属性是一个类类型，那么在定位实现的方法时，会先搜索这个类以及它的祖先类，然后再搜索当前类（也就是定义属性的类）以及它的祖先类。所以，可以在属性指定的类中实现某些方法，而另一些方法在当前类实现。可以象平常一样使用方法解析子句来避免含糊的声明，或者（只是）使用一个特别的方法声明。一个接口不能委托给多个类类型的属性实现。比如

```
type
  IMyInterface = interface
    procedure P1;
    procedure P2;
  end;
  TMyImplClass = class
    procedure P1;
    procedure P2;
  end;
  TMyClass = class(TInterfacedObject, IMyInterface)
    FMyImplClass: TMyImplClass;
    property MyImplClass: TMyImplClass read FMyImplClass implements
IMyInterface;
    procedure IMyInterface.P1 = MyP1;
    procedure MyP1;
```

```
  end;
procedure TMyImplClass.P1;
  ...
procedure TMyImplClass.P2;
  ...
procedure TMyClass.MyP1;
  ...
var
  MyClass: TMyClass;
  MyInterface: IMyInterface;
begin
  MyClass := TMyClass.Create;
  MyClass.FMyImplClass := TMyImplClass.Create;
  MyInterface := MyClass;
  MyInterface.P1;              // 调用 TMyClass.MyP1;
  MyInterface.P2;              // 调用 TImplClass.P2;
end;
```

Interface references（接口引用）

Interface references（接口引用）

如果你声明一个接口类型的变量，则它可以引用任何实现这个接口的类实例。这样的变量使你可以调用接口的方法，而不必在编译时知道接口是在哪里实现的。但要注意以下限制：

- 使用接口类型的表达式只能访问接口定义的方法和属性，不能访问实现类的其它成员；
- 一个接口类型的表达式不能引用实现了它的派生接口的类实例，除非这个类（或它继承的类）还明确实现了此祖先接口。

比如，

```
type
  IAncestor = interface
  end;
  IDescendant = interface(IAncestor)
    procedure P1;
  end;
  TSomething = class(TInterfacedObject, IDescendant)
    procedure P1;
    procedure P2;
  end;
  ...
var
  D: IDescendant;
  A: IAncestor;
begin
  D := TSomething.Create;      // 工作正常！
  A := TSomething.Create;      // 出错
```

```
    D.P1;                              // 工作正常!
    D.P2;                              // 出错
  end;
```

在这个例子中，

- A 被声明为 IAncestor 类型的变量，因为 TSomething 声明实现的接口中没有列出 IAncestor，TSomething 类型的实例不能赋给 A。但如果改变 TSomething 的声明为

  ```
  TSomething = class(TInterfacedObject, IAncestor, IDescendant)
   ...
  ```

  那么第一个错误语句将变得可用（A := TSomething.Create;）
- D 被声明为 IDescendant 类型的变量，虽然它可以引用 Tsomething 类型的实例，但我们不能用它访问 TSomething 的 P2 方法，因为它不是 IDescendant 接口的方法。但如果改变 D 的声明为

  ```
  D: TSomething;
  ```

  则第二个错误语句将变为可用。（D.P2;）

接口引用通过引用计数进行管理，它依赖于从 IInterface 继承的_AddRef 和_Release 方法。若一个对象只通过接口来引用，我们没必要手动销毁它，当它最后的引用超出范围时，它会自动销毁。

全局类型的接口变量能只被初始化为 **nil**。

要判断一个接口类型的表达式是否引用了一个对象，通过标准函数 Assigned 来完成。

Interface assignment-compatibility（接口赋值兼容性）

一个类和它实现的任何接口是赋值兼容的，一个接口和它的任何祖先接口是赋值兼容的。**nil** 可以被赋给任何接口类型的变量。

一个接口类型的表达式可以被赋予一个变体类型（Variant）：若接口类型是 IDispatch 或它的后代，则 Variant 变量的类型码是 varDispatch，否则为 varUnknown。

类型码为 varEmpty、varUnknown 或者 varDispatch 的 Variant 变量，可以赋给 IInterface 类型的变量；类型码为 varEmpty 或 varDispatch 的 Variant 变量，可以赋给 IDispatch 类型的变量。

Interface typecasts（接口类型转换）

对于变量和值类型转换（variable and value typecast），接口类型和类类型遵循同样的原则。若一个类实现了某个接口，则类类型可以转换为这个接口类型，比如 IMyInterface(SomeObject)。

一个接口类型的表达式可以转换为变体类型。如果接口类型是 IDispatch 或者它的后代，则变量的类型码是 varDispatch，否则为 varUnknown。

类型码为 varEmpty、varUnknown 或 varDispatch 的 Variant 变量，可以转换为 IInterface 接口类型；类型码为 varEmpty 或 varDispatch 的 Variant 变量，可以转换为 IDispatch 接口类型。

Interface querying（接口查询）

你可以使用 as 运算符进行受检查的接口转换，我们称它为接口查询。它从一个类引用转换为接口类型，或从接口引用转换为另一种接口类型，它基于实际的（运行时）对象类型。接口查询有如下格式

object **as** *interface*

这里，*object* 是一个接口类型的表达式，或者是一个变体类型，或者是实现了某个接口的类实例，*interface* 是任何一个声明了 GUID 的接口。（能查询的接口必须声明 GUID）

如果 *object* 是 **nil**，则它返回 **nil**；否则，它传递 *interface* 接口的 GUID 到 *object* 的 QueryInterface 方法：

若 QueryInterface 不是返回 0，它引发一个异常；若 QueryInterface 返回 0（表示 *object* 的类实现了接口），则接口查询（**as** 语句）返回 *object* 的接口引用。

Automation objects（自动化对象）

Automation objects (自动化对象)

若对象所属的类实现了 IDispatch 接口（在 System 单元声明），则此对象是一个自动化对象。自动化对象只适用于 Windows。

Dispatch interface types (派遣接口类型)

派遣接口类型定义了一个自动化对象的方法和属性，它们通过 IDispatch 接口来实现。调用派遣接口的方法是通过在运行时调用 IDispatch 接口的 Invoke 方法来实现的，a class cannot implement a dispatch interface。

派遣接口声明具有如下格式：

```
type interfaceName = dispinterface
  ['{GUID}']
  memberList
end;
```

这里，['{GUID}']是可选的，memberList 包括属性和方法声明。派遣接口和普通接口有类似的声明，但它们不能指定一个祖先。比如，

```
type
  IStringsDisp = dispinterface
    ['{EE05DFE2-5549-11D0-9EA9-0020AF3D82DA}']
    property ControlDefault[Index: Integer]: OleVariant dispid 0; default;
    function Count: Integer; dispid 1;
    property Item[Index: Integer]: OleVariant dispid 2;
    procedure Remove(Index: Integer); dispid 3;
    procedure Clear; dispid 4;
    function Add(Item: OleVariant): Integer; dispid 5;
    function _NewEnum: IUnknown; dispid -4;
  end;
```

派遣接口的方法

派遣接口的方法是一种函数原型，由 IDispatch 接口的 Invoke 方法使用。（Methods of a dispatch interface are prototypes for calls to the Invoke method of the underlying IDispatch implementation）要给一个方法指定派遣号码（ID），声明方法时包含 **dispid** 指示字，并在它的后面跟一个整数常量。如果指定的整数已经用过了，则产生错误。

除了 **dispid**，在派遣接口中声明的方法不能使用其它指示字，它的参数以及返回值必须属于自动化类型，也就是说，必须是 Byte、Currency、Real、Double、Longint、Integer、Single、Smallint、AnsiString、WideString、

派遣接口的属性

派遣接口的属性不能包含访问说明符，它们可以声明为 **readonly** 或 **writeonly**。要给一个属性指定派遣

号码，声明时包含 **dispid** 指示字，并在它的后面跟一个整数常量。如果指定的整数已经用过了，则产生错误。数组属性能被声明为 **default**，其它指示字在声明派遣接口的属性时是不允许的。

Accessing automation objects (访问自动化对象)

使用变体类型来访问自动化对象。当一个 Variant 变量引用一个自动化对象时，可以通过它调用对象的方法以及读或写它的属性。要做到这些，你必须在单元、程序或库的 **uses** 子句中包含 ComObj（单元）。调用自动化对象的方法是在运行时绑定的，不需要事先声明这些方法，但调用的有效性在编译时不进行检查。

下面的例子演示了调用自动化方法。CreateOleObject 函数（在 ComObj 定义）返回一个自动化对象的 IDispatch 引用，它和 Variant 变量 Word 是赋值兼容的。

```
var
  Word: Variant;
begin
  Word := CreateOleObject('Word.Basic');
  Word.FileNew('Normal');
  Word.Insert('This is the first line'#13);
  Word.Insert('This is the second line'#13);
  Word.FileSaveAs('c:\temp\test.txt', 3);
end;
```

你可以给自动化方法传递接口类型的参数。（？）

使用元素类型为 varByte 的 Variant 数组，是自动化控制器和服务器之间交换二进制数据的首选方法。使用这样的数组不需要转换它们的数据，并且能使用 VarArrayLock 和 VarArrayUnlock 例程进行有效的操作。

调用自动化方法的语法

调用自动化对象的方法或访问它们的属性，与普通的方法调用和属性访问是类似的，但是，调用自动化方法既可以使用定位参数（positional），也可以使用命名（named）参数（但有些自动化服务器不支持命名参数）。

定位参数就是一个表达式，命名参数包括一个参数标志符，后面跟:=，再跟一个表达式。在调用方法时，定位参数必须在所有的命名参数之前，命名参数可以使用任意顺序。

一些自动化服务器允许你在调用方法时省略参数，而使用它们的默认值。比如，

```
Word.FileSaveAs('test.doc');
Word.FileSaveAs('test.doc', 6);
Word.FileSaveAs('test.doc',,,'secret');
Word.FileSaveAs('test.doc', Password := 'secret');
Word.FileSaveAs(Password := 'secret', Name := 'test.doc');
```

自动化方法的参数可以是整数、实数、字符串、布尔和变体类型。若参数表达式只是一个变量引用，并且变量类型属于 Byte、Smallint、Integer、Single、Double、Currency、TDateTime、AnsiString、WordBool 或 Variant，参数以引用形式传递（传址）；若参数表达式不是上述类型，或不止是一个变量，参数以数值方式传递（传值）。当使用传址方式调用一个以传值方式传递参数的方法时，会导致 COM 取得（/取回）引用参数的值；而当使用传值方式调用一个以传址方式传递参数的方法时，会导致错误。

Dual interfaces (双重接口)

双重接口既支持编译时绑定，也支持通过自动化动态绑定（运行时）。双重接口必须从 **IDispatch** 接口派

生。

双重接口的所有方法（除了从 **IInterface** 和 **IDispatch** 继承的方法）必须使用 **safecall** 调用约定，并且方法的参数和返回值必须是自动化类型。（自动化类型包括 Byte、Currency、Real、Double、Real48、Integer、Single、Smallint、AnsiString、TdateTime、Variant、OleVariant 和 WordBool）

Memory management（内存管理）

Memory management: Overview（概述）

注意：Linux 使用 glibc 函数，比如 *malloc*，进行内存管理。要了解更多信息，请参考 Linux 系统关于 *malloc* 的帮助文件。

在 Windows 系统下，内存管理器负责程序中所有的动态内存分配和回收。*New*、*Dispose*、*GetMem*、*ReallocMem* 和 *FreeMem* 标准过程使用内存管理器，所有的对象和长字符串也通过内存管理器来进行分配。

在 Windows 下，对于面向对象的程序和处理字符数据的程序，典型情况下，它们需要分配大量的较小或中等大小的内存块，内存管理器对这种情况进行了优化。而其它的内存管理器，象 *GlobalAlloc* 和 *LocalAlloc* 的实现方式以及 Windows 支持的私有堆，在这种情形下性能并不好，当直接使用时，会降低程序速度。

为确保最好的性能，内存管理器直接和 Win32 虚拟内存 API（*VirtualAlloc* 和 *VirtualFree* 函数）打交道。内存管理器从操作系统中保留（reserve）地址空间时，以 1MB 为一节（单位）；当需要提交（commit）物理内存时，以 16KB 的幅度进行。当释放内存和地址空间时，也是以 16KB 和 1MB 为单位的。对于更小的（内存）块，在已提交的内存中进行再分配。

内存管理器块总是以 4 个字节进行对齐，并总是拥有一个 4 字节的头，这里包含内存块的大小及其它信息位。这意味着，内存管理器块总是以双字的形式优化排列，以保证定位内存块时 CPU 的效能发挥得最好。

内存管理器维护着两个状态变量：*AllocMemCount* 和 *AllocMemSize*，它们保存着当前分配的内存块数目、以及这些内存块的总容量。在调试时，应用程序可以利用这些变量来显示状态信息。

System 单元提供了两个过程：*GetMemoryManager* 和 *SetMemoryManager*，它们允许程序拦截底层的内存管理器调用。System 单元还提供了 *GetHeapStatus* 函数，它返回一个包含内存管理器详细状态信息的记录结构。

Variables（变量）

全局变量在程序的数据段分配，并且在程序运行期间一直存在；局部变量（在过程或函数内声明）存在于程序的堆栈中，每次调用过程或函数，局部变量进行分配，而调用结束后，局部变量被清除。编译器优化可能提前消除变量（比如使用寄存器）。

注意：在 Linux 下，堆栈大小只能由环境设置。

在 Windows 下，一个程序的堆栈由两个值定义：堆栈的最小值和最大值。这两个值受编译器指示字 **$MINSTACKSIZE** 和 **$MAXSTACKSIZE** 所控制，它们的缺省值分别是 16,384（16K）和 1,048,576（1M）。程序保证拥有最小容量的堆栈空间，且不允许超过堆栈的最大值。如果空闲内存不能保证最小的堆栈需求，Windows 在启动程序时会报告出错。

如果程序需要的堆栈容量超过最小值，它会以 4K 的幅度自动增加。如果分配额外的堆栈失败，可能是没有足够的空闲内存，或者堆栈容量达到了它所允许的最大值。此时，将引发 EStackOverflow 异

常（堆栈溢出检查完全是自动的。编译器指示字 **$S** 原先是控制堆栈溢出检查的，保留它是为了向后兼容性）。

在 Windows 或 Linux 下，由 *GetMem* 或 *New* 过程创建的动态变量是在堆中分配的，除非使用 *FreeMem* 或 *Dispose* 命令进行释放，否则它们将一直存在。

长字符串、宽字符串、动态数组、variant 以及接口在堆中进行分配，但它们的内存是自动管理的。

Integer types（整数类型）

整型变量的格式取决于于它的最小值和最大值边界：

- 若边界都介于 -128..127（*Shortint*），存储为有符号字节（signed byte）
- 若边界都介于 0..255 （*Byte*），存储为无符号字节（unsigned byte）
- 若边界都介于 -32768..32767（*Smallint*），存储为有符号字（signed word）
- 若边界都介于 0..65535（*Word*），存储为无符号字（unsigned word）
- 若边界都介于 -2147483648..2147483647（*Longint*）， 存储为有符号双字（signed double word）
- 若边界都介于 0..4294967295（*Longword*），存储为无符号双字（unsigned double word）
- 否则（*Int64*），存储为有符号四字（signed quadruple word）

Character types（字符类型）

Char、*AnsiChar* 或 *Char* 的子界类型存储为一个无符号字节，*WideChar* 存储为一个无符号字。

Boolean types（布尔类型）

Boolean 类型存储为 *Byte*，*ByteBool* 也存储为 *Byte*，*WordBool* 存储为 *Word*，*LongBool* 存储为 *Longint*。

Boolean 类型把 0 作为 *False*，1 作为 *True*；*ByteBool*、*WordBool* 和 *LongBool* 把 0 作为 *False*，非 0 作为 *True*。

Enumerated types（枚举类型）

若枚举类型的值不超过 256 个，并且在 **{$Z1}** 状态（默认）下声明，它被存储为一个无符号字节；若枚举类型的值超过 256 个，或在 **{$Z2}** 状态下声明，它被存储为一个无符号字；若枚举类型在 **{$Z4}** 状态下声明，它被存储为无符号双字。

Real types（实数类型）

The real types store the binary representation of a sign (+ or –), an exponent, and a significand. A real value has the form

+/?significand * 2^exponent

where the significand has a single bit to the left of the binary decimal point. (That is, 0 <= significand < 2.)
In the figures that follow, the most significant bit is always on the left and the least significant bit on the right. The numbers at the top indicate the width (in bits) of each field, with the leftmost items stored at the highest addresses. For example, for a Real48 value, e is stored in the first byte, f in the following five bytes, and s in the most significant bit of the last byte.

Pointer types（指针类型）

指针类型有 4 个字节，作为一个 32 位地址。指针值 **nil** 存储为 0。

Short string types（短字符串类型）

短字符串占用的字节数是它的最大长度加 1，第一个字节存储字符串的当前（动态）长度，剩下的字节存储字符串的字符。

存储（字符串）长度的字节和字符都被看作无符号值。最大字符串长度是 255 个字符加上 1 个保存长度的字节（**string**[255]）。

Long string types（长字符串）

长字符串变量是一个占用 4 字节内存的指针，指向一个动态分配的字符串。当字符串变量为空时（字符串长度为零），指针为 **nil** 并且不分配动态内存。对一个非空值，字符串指针指向一个动态分配的内存块，这个内存块包含了字符串的实际内容，并且还有一个 32 位的值来指示字符串的长度，同时还包括一个 32 位的引用计数。下面的表格说明了内存块的分配情况。

偏移量	内容
−8	32 位引用计数
−4	长度（字节数）
0..*Length*−1	字符串
Length	NULL 字符（#0）

内存块末尾的 NULL 字符由编译器和内置的字符串处理例程自动维护，这使得长字符串能直接转换为一个零结尾字符串。

对字符串常量和文字串（literal），编译器象动态分配时一样为其分配内存，但把它的引用计数设置为−1。当把字符串常量赋给一个字符串变量时，字符串指针指向为常量分配的内存块。当字符串引用记数为−1 时，内置的字符串处理例程知道不能去修改它。

Wide string types（宽字符串）

在 Windows 下，宽字符串变量是一个占用 4 字节内存的指针，指向一个动态分配的字符串。当字符串变量为空时（字符串长度为零），指针为 **nil** 并且不分配动态内存。对一个非空值，字符串指针指向一个动态分配的内存块，这个内存块包含了字符串的实际内容，并且还有一个 32 位的值来指示字符串的长度（没有引用计数）。下面的表格说明了内存块的分配情况。

偏移量	内容
−4	32 位长度指示器（字节数）
0..*Length* −1	字符串
Length	NULL 字符（#0）

字符串的长度以字节为单位，所以，它是字符串所包含的字符数目的两倍。

内存块末尾的 NULL 字符由编译器和内置的字符串处理例程自动维护，这使得宽字符串能直接转换为一个零结尾字符串。

Set types（集合类型）

集合可看作是由位（bit）组成的数组，每个位指明一个元素是否在集合中。一个集合最多有 256 个元素，所以，一个集合占用的空间不会超过 32 个字节。对一个特定的集合，它占用的字节数等于

(*Max* **div** 8) - (*Min* **div** 8) + 1

这里，*Max* 和 *Min* 是集合基础类型的上下边界。集合中一个特定元素 *E* 所在的字节（序号）是

(E **div** 8) - (*Min* **div** 8)

在这个字节中，它对应的位（序号）是

E **mod** 8

这里，*E* 表示元素的序数值。编译器尽可能把集合存储在 CPU 寄存器中，但若它的大小比普通 *Integer* 类型大，或在程序的代码中使用了集合的地址，它总是被存储在内存中。

Static array types（静态数组）

静态数组是由它的元素按顺序构成的序列，拥有最小索引的元素在内存块的底端。对多维数组来说，最右边的一维先发生变化。

Dynamic array types（动态数组）

动态数组变量是一个占用 4 字节内存的指针，指向动态分配内存的数组。当变量为空（未初始化）或存储一个长度为 0 的数组时，指针为 **nil**，并且不会为数组分配内存。对一个非空数组，指针指向一个动态分配的内存块，这个内存块包含了数组的值，并且还有一个 32 位的长度指示、以及一个 32 位的引用计数。下面的表格说明了动态数组的内存分配情况。

偏移量	内容
−8	32 位引用计数
−4	32 位长度指示（元素个数）
0..*Length* * (size of element) − 1	数组的元素

Record types（记录类型）

When a record type is declared in the {$A+} state (the default), and when the declaration does not include a

packed modifier, the type is an unpacked record type, and the fields of the record are aligned for efficient access by the CPU. The alignment is controlled by the type of each field. Every data type has an inherent alignment, which is automatically computed by the compiler. The alignment can be 1, 2, 4, or 8, and represents the byte boundary that a value of the type must be stored on to provide the most efficient access. The table below lists the alignments for all data types.

当一个记录类型在**{$A+}**状态下声明，并且没有使用 **packed** 修饰字时，这是一个未压缩的（unpacked）记录类型。此时，记录的各字段被优化排列，以使 CPU 更有效地进行访问。排列规则由字段的类型控制，每一种数据类型有固定的排列规则，并且由编译器自动计算。排列规则可以是 1，2，4 或者 8，它表示每一个类型被存储的字节边界，以得到最优化的访问。下表列出了所有数据类型的排列规则。

类型	字节对齐
有序类型	类型的大小（1、2、4 或者 8）
实数	2 for Real48, 4 for Single, 8 for Double and Extended
短字符串	1
数组	same as the element type of the array.
记录	the largest alignment of the fields in the record
集合	size of the type if 1, 2, or 4, otherwise 1
其它所有类型	4

To ensure proper alignment of the fields in an unpacked record type, the compiler inserts an unused byte before fields with an alignment of 2, and up to three unused bytes before fields with an alignment of 4, if required. Finally, the compiler rounds the total size of the record upward to the byte boundary specified by the largest alignment of any of the fields.

当在**{$A–}**状态下声明记录类型，或者当声明包含 **packed** 修饰符时，记录的字段没有被优化排列，它们被连续存放。这样一个压缩（packed）记录，它的大小是所有字段的大小之和。因为数据排列会改变，所以，当把记录结构写入磁盘，或者通过内存把记录结构传递给由不同版本的编译器编译的模块时，使用压缩记录是个好主意。

File types（文件类型）

文件类型表示为记录，类型文件和无类型文件占用 332 字节，它们的布局如下：

```
type
  TFileRec = packed record
    Handle: Integer;
    Mode: word;
    Flags: word;
    case Byte of
      0: (RecSize: Cardinal);
      1: (BufSize: Cardinal;
          BufPos: Cardinal;
          BufEnd: Cardinal;
          BufPtr: PChar;
          OpenFunc: Pointer;
          InOutFunc: Pointer;
          FlushFunc: Pointer;
          CloseFunc: Pointer;
```

```
            UserData: array[1..32] of Byte;
            Name: array[0..259] of Char; );
    end;
```

文本文件占用 460 字节，它的布局如下：

```
  type
    TTextBuf = array[0..127] of Char;
    TTextRec = packed record
      Handle: Integer;
      Mode: word;
      Flags: word;
      BufSize: Cardinal;
      BufPos: Cardinal;
      BufEnd: Cardinal;
      BufPtr: PChar;
      OpenFunc: Pointer;
      InOutFunc: Pointer;
      FlushFunc: Pointer;
      CloseFunc: Pointer;
      UserData: array[1..32] of Byte;
      Name: array[0..259] of Char;
      Buffer: TTextBuf;
    end;
```

Handle 保存文件的句柄（当文件打开时）。

Mode 字段能被赋予下列值之一

```
  const
    fmClosed = $D7B0;
    fmInput   = $D7B1;
    fmOutput = $D7B2;
    fmInOut   = $D7B3;
```

这里，fmClosed 表示文件已经被关闭，fmInput 和 fmOutput 表示打开了（reset）一个文本文件（fmInput）或创建并打开了（rewritten）一个新文本文件（fmOutput），fmInOut 表示打开或创建了一个类型或无类型文件。任何其它的值表示文件变量还没有被赋值（因此没有被初始化）。

用户定义的写入例程使用 UserData 字段来存储数据。

Name 字段保存文件名，它是一个以 0 字符（#0）结尾的字符序列。

对类型文件和无类型文件，RecSize 包含记录的长度（字节），Private 字段（？）没有使用但是保留的。

对文本文件，BufPtr 是指向一个缓冲区的指针，缓冲区的大小由 BufSize 说明，BufPos 是缓冲区中下一个要读写的字符的索引，BufEnd 是缓冲区中有效字符的数目。OpenFunc、InOutFunc、FlushFunc 和 CloseFunc 是指向 I/O 例程的指针，请参考 Device functions。Flags 决定了换行风格，像下面所示：

bit 0 clear（清除 0 位）	LF line breaks（换行）
bit 0 set（设置 0 位）	CRLF line breaks（回车换行）

Flags 其它所有的位被保留以备将来使用。请参考 DefaultTextLineBreakStyle 和 SetLineBreakStyle。

Procedural types（过程类型）

过程指针存储为一个指向过程或函数入口点的 32 位指针；方法指针存储为一个指向方法入口点的 32 位指针，后面还跟一个指向对象的 32 位指针。

Class types（类类型）

类类型的值存储为一个 32 位的指针，它指向一个类的实例，我们称它为对象。对象的内部数据结构就象一个记录，它的字段以声明的顺序进行存储，就象一系列连续的变量。对象的字段总是被优化排列，就象未压缩的记录类型。从祖先类继承下来的所有字段存储在新声明的字段之前。

每个对象的前 4 个字节是一个指针，它指向类的虚方法表（VMT）。每个类有一个虚方法表，而不是每个对象有一个。不同的类类型，不管多么相似，都不会共用 VMT。VMT 由编译器自动创建，不能由程序直接操纵。指向 VMT 的指针，是由对象的构造函数自动存储的，也不能由程序直接操纵。

VMT 的布局如下表所示。在正偏移方向，VMT 包含一个由 32 位方法指针构成的列表，每个指针对应于类中用户定义的一个虚方法，方法指针的顺序和声明的顺序相一致，每个指针包含相应的虚方法的入口地址。这种布局和 C++的虚表（v-table）以及 COM 兼容。在负偏移方向，VMT 也包含很多字段，它们完成 Object Pascal 的内部实现。应用程序应该使用 Tobject 定义的方法来查询这一信息，因为在将来这种实现方式有可能改变。

偏移量	类型	描述
-76	Pointer	指向虚方法表的指针（或 **nil**）
-72	Pointer	指向接口表的指针（或 **nil**）
-68	Pointer	指向自动化信息表的指针（或 **nil**）
-64	Pointer	指向实例初始化表的指针（或 **nil**）
-60	Pointer	指向类型信息表的指针（或 **nil**）
-56	Pointer	指向字段（定义）表的指针（或 **nil**）
-52	Pointer	指向方法（定义）表的指针（或 **nil**）
-48	Pointer	指向动态方法表的指针（或 **nil**）
-44	Pointer	指向包含类名的短字符串的指针
-40	Cardinal	实例的字节大小
-36	Pointer	指向祖先类的指针的指针（或 **nil**）（指针的指针）
-32	Pointer	指向 SafecallException 方法入口指针的指针（或 **nil**）（指针的指针）
-28	Pointer	AfterConstruction 方法入口（指针）
-24	Pointer	BeforeDestruction 方法入口（指针）
-20	Pointer	Dispatch 方法入口（指针）
-16	Pointer	DefaultHandler 方法入口（指针）
-12	Pointer	NewInstance 方法入口（指针）
-8	Pointer	FreeInstance 方法入口（指针）
-4	Pointer	析构函数 Destroy 的入口地址（指针）
0	Pointer	用户自定义的第一个虚方法入口（指针）
4	Pointer	用户自定义的第二个虚方法入口（指针）
...	...	...

Class reference types（类引用）

类引用（值）存储为一个 32 位指针，它指向一个类的虚方法表（VMT）。

Variant types（Variant 类型）

variant 存储为一个 16 字节的记录，它包含类型码，以及类型码指明的数据类型的值（或值的引用）。System

和 Variants 单元定义了 variant 常量和类型。

TVarData 类型表示一个 variant 变量的内部结构（在 Windows 下，它和 COM 以及 Win32 API 中使用的 variant 是相同的），它能用来对 variant 变量进行类型转换，以便访问变量的内部结构。

TVarData 记录的 VType 字段包含了类型码，它存储在较低的 12 个位中（每个位由 *varTypeMask* 常量定义）。并且，可能设置 varArray 位用于表明 variant 是数组，也可能设置 varByRef 位用于表明 variant 存储的是值的引用而不是值本身。

TVarData 的 Reserved1、Reserved2 和 Reserved3 字段没有使用。

TVarData 记录剩余 8 个字节的内容取决于 VType 字段。若既没有设置 varArray 位也没有设置 varByRef 位，则它包含指定类型的值。

若设置了 varArray 位，variant 包含一个指向 TVarArray 结构的指针，TVarArray 定义了数组，每个数组元素的类型由 Vtype 字段的 *varTypeMask* 位指明。

若设置了 varByRef 位，variant 包含一个值的引用，它的类型由 VType 字段的 varTypeMask 和 varArray 位指明。

varString 类型码是私有的，包含此类型的 variant 不能传给非 Delphi 函数。在 Windows 下，当把 variant 作为参数传给外部函数时，Delphi（的自动化支持）自动把 varString 转换为 varOleStr 类型。

在 Linux 下，不支持 VT_decimal。

Program control（程序控制）

Program control: Overview（概述）

这一部分解释参数、和函数结果如何存储和传输。最后介绍 exit 过程。

Parameters and function results（参数和函数结果）

Parameters and function results: Overview（概述）

有几个因素决定参数、函数结果如何被处理，它们包括调用约定、参数语义以及传递值的类型和大小。

Parameter passing（参数传递）

参数是通过 CPU 寄存器或栈传递给过程或函数的，这取决于例程的调用约定。要了解调用约定的信息，请参考 Calling conventions。
变量参数（**var**）总是通过引用传递（作为 32 位指针），它指向（变量参数的）实际存储位置。
值传递和常量参数（**const**）可通过值传递，也可能通过引用传递，这取决于参数的类型和大小：

- 一个有序类型的参数通过 8 位、16 位、32 位或 64 位值进行传递，它们的格式和相同类型的变量一致。
- 一个实数参数总是被传递到栈中。一个 Single 类型的参数占用 4 字节；Double、Comp 或 Currency 类型占用 8 字节；Real48 也占用 8 字节，它的值存储在较低的 6 个字节中；Extended 类型占用 12 字节，它的值存储在较低的 10 个字节中。
- 一个短字符串类型的参数，作为一个指向它的 32 位指针传递。
- 一个长字符串或动态数组类型的参数，也作为 32 位指针，它指向分配的内存块。值 **nil** 作为一个空（长）串传递。
- 一个指针、类、类引用或过程（指针）类型的参数作为 32 位指针传递。
- 一个方法指针作为两个 32 位指针被传递到栈中。实例（对象）指针在方法指针之前被压入栈，所以方法指针占据较低的地址。
- 在 **register** 和 **pascal** 调用约定下，一个 Variant 类型参数被作为 32 位指针传递。
- 1、2 或 4 字节的集合、记录和静态数组被当作 8 位、16 位和 32 位值传递。较大的集合、记录和静态数组被作为 32 位指针。一个例外是，当使用 **cdecl**、**stdcall** 或 **safecall** 调用约定时，一个记录类型总是直接传给栈，此时，记录的大小被圆整位双字（4 字节的倍数）。
- 一个开放数组类型的参数作为 2 个 32 位值传递。第一个是指向数组的指针，第 2 个是数组的元素个数减 1（也就是数组的最大下标）。

当两个参数被传递到栈时，每个参数占用 4 字节的倍数。对 8 位或 16 位参数来说，即使它们只占用 1 个字节或 1 个字，它也作为双字传递。双字中未用的部分没有定义。
在 **pascal**、**cdecl**、**stdcall** 和 **safecall** 约定下，所有的参数被传递给栈。在 pascal 约定下，参数按它们声明的顺序（从左到右）被压入栈，所以，第一个参数在最高地址，最后一个参数在最低地址。在 **cdecl**、**stdcall** 和 **safecall** 约定下，参数以声明的相反顺序（从右到左）被压入栈，所以，第一个参数在最低位置，而最后一个在最高位置。
在 **register** 约定下，最多有 3 个参数可通过 CPU 寄存器传递，其余（若有的话）参数被传递到栈。此时，

参数以声明的顺序（和 **pascal** 相同）被传递，前 3 个有资格的参数分别使用 EAX、EDX 和 ECX 寄存器。实数、方法指针、Variant、Int64 和结构类型不能作为寄存器参数（没有资格），其它类型都可以。若具有资格的参数多于 3 个，前 3 个被选用，其余的以声明的顺序传给栈。比如，下面的声明中

```
procedure Test(A: Integer; var B: Char; C: Double; const D: string; E: Pointer);
```

A 作为 32 位整数传给 EAX，B 作为字符指针（var 参数）传给 EDX，D 作为指针传给 ECX；C 和 E 作为双字和指针被压入栈，和它们声明的顺序一样。

寄存器保存约定

过程和函数必须保留 EBX、ESI、EDI 和 EBP 寄存器，但可以修改 EAX、EDX 和 ECX。当在汇编语言中实现构造和销毁时，保证预留 DL 寄存器。过程和函数被调用时，是假定 CPU 的 direction 标志是清除的（对应于 CLD 指令），并且返回时，direction 标志也必须是清除的。

Function results（函数结果）

以下约定适用于函数的返回值：

- 可能的话，有序类型通过寄存器返回值：字节通过 AL 返回，字通过 AX 返回，双字通过 EAX 返回。
- 实数类型的返回值在浮点协处理器的栈顶寄存器（top-of-stack register，ST(0)）。对于 Currency 类型的返回值，ST（0）中的值被乘以 10000。比如，Currency 值 1.234 在 ST（0）中的值为 12340。
- 对字符串、动态数组、方法指针、Variant、或 Int64 类型的返回值，就像函数在其它参数的后面额外声明了一个 **var** 参数。换句话说，是函数调用者传递一个额外的 32 位指针，它指向的变量用来返回结果。
- 指针、类、类引用和过程指针类型，结果通过 EAX 返回。
- 对静态数组、记录和集合类型，若结果占用 1 个字节，它通过 AL 返回；若结果占用 2 个字节，它通过 AX 返回；若结果占用 4 个字节，它通过 EAX 返回。否则（结果超过 4 个字节），结果通过一个额外的 **var** 参数返回，它在所有声明的参数的后边。

Method calls（方法调用）

方法和其它普通过程（和函数）使用相同的调用约定，除了每个方法有一个隐含的参数 Self，这是一个实例或类的引用。Self 参数作为 32 位指针传递。

在 **register** 调用约定下，Self 就像在所有其它参数的前面声明，所以，它总是通过 EAX 寄存器传递。

在 **pascal** 调用约定下，Self 就像在所有其它参数的后面声明（有时还要包括返回函数值的额外的 var 参数），所以，它最后被压入栈，所在的地址比其它参数要低。

在 **cdecl**、**stdcall** 和 **safecall** 调用约定下，Self 就像在所有其它参数的前面声明，但却在作为函数返回值的额外的 **var** 参数之后（如果有的话），所以，除了额外的 **var** 参数，它最后一个被压入栈。

Constructors and destructors（构造函数和析构函数）

Constructor 和 destructor 与其它方法使用相同的调用约定，除了传递一个附加的布尔类型的标志，它指示 Constructor 和 destructor 的调用环境。

对 Constructor 来说，若布尔标志是 False，则表明是通过一个（对象）实例进行调用或使用 **inherited** 关键字，此时，它就像一个普通方法；若标志值是 True，则表明是通过一个类引用进行调用，此时，constructor 创建一个类实例（given by Self，通过 Self 给出），并在 EAX 中给出新对象的引用。

对 Destructor 来说，若布尔标志是 False，则表明是使用 **inherited** 进行调用，此时，它就像一个普通的方法；若标志值是 True，则表明是通过一个（对象）实例进行调用，此时，destructor 在返回前释放实例（given by Self，实例通过 Self 给出）。

这个标志参数就像在其它所有参数之前声明。在 **register** 调用约定下，它通过 DL 寄存器传递；在 **pascal** 约定下，它在其它参数之前被压入栈；在 **cdecl**、**stdcall** 和 **safecall** 约定下，它在其它参数之后、在 Self 之前被压入栈。

因为 DL 寄存器指示 constructor 或 destructor 是否在调用栈的最外面（outermost），所以在退出之前你必须恢复 DL 的值，这样 BeforeDestruction 和 AfterConstruction 才能被正确调用。

Exit procedures（结束过程，非 Exit 过程）

结束过程使某些动作（比如更新文件和关闭文件）在程序结束之前被执行。ExitProc 指针变量允许你‘安装’一个结束过程，它在程序结束时总是被执行：不管程序是正常结束，还是调用 Halt，还是产生了运行时错误而退出。

注意：要对程序的退出采取行动时，推荐使用结束化部分（finalization section），而不是结束过程。结束过程只对执行文件、共享对象（Linux）或 DLL（Windows）是可用的，包则必须使用结束化部分进行处理。所有结束过程在结束化部分（finalization section）之前被执行。

单元可以像程序一样安装结束过程。一个单元可在初始化代码中‘安装’结束过程，依靠它来关闭文件或执行清除任务。

当正确实现时，一个结束过程是结束过程链中的一部分。结束过程和安装过程的顺序相反，这样保证了一个单元的结束过程不会在另一个依赖它的单元之前被执行。要保证链的完整无缺，在把 ExitProc 指向自己的过程之前必须保存它的内容，而且，在你的结束过程中，第一个命令就是把 ExitProc 恢复为原来保存的值。

下面的代码演示了一个结束过程的实现结构：

```
var
  ExitSave: Pointer;

procedure MyExit;
begin
  ExitProc := ExitSave;  // 总是先恢复原来的值
  …
end;

begin
  ExitSave := ExitProc;
  ExitProc := @MyExit;
  …
end.
```

在程序入口，代码先保存 ExitProc 中的内容到 ExitSave，然后‘安装’MyExit 过程。当作为结束过程的一部分被调用时，MyExit 首先要做的事就是重新‘安装’以前的结束过程。

运行库（Delphi 的运行库？）中的结束例程会一直调用结束过程，直到 ExitProc 变成 **nil**。为避免无限制循环，在每个调用之前，ExitProc 被设置为 **nil**，所以，只有在当前的结束过程中给 ExitProc 赋一个地址，下一个结束过程才被调用。若在一个过程中发生了错误，它就不会再调用。

一个结束过程，通过检测 ExitCode（整数）和 ErrorAddr（指针）变量的值，可得知程序结束的原因。在正常结束的情况下，ExitCode 值为 0，ErrorAddr 为 **nil**；当通过调用 Halt 结束时，ExitCode 值为传给 Halt 的参数，ErrorAddr 为 **nil**；若发生运行时错误，ExitCode 包含错误码，ErrorAddr 包含无效指令的地址。

最后一个结束过程（通过运行库安装）关闭输入和输出文件。如果 ErrorAddr 不为 **nil**，它输出一个运行时错误信息。要输出你自己的运行时错误信息，‘安装’一个结束过程来检测 ErrorAddr，若它不为 **nil**，就输出一个信息。在返回前，把 ErrorAddr 设置为 **nil**，这样，其它结束过程就不会重新报告错误信息。一旦运行库调用了所有的结束过程，它返回操作系统，把存储在 ExitCode 中的值作为返回结果。

Inline assembler code

Inline assembler code: Overview

The asm statement

Assembler statement syntax

Assembler statement syntax

Labels

Instruction opcodes

Assembler directives

Operands

Expressions

Expressions: Overview

Differencces between Object Pascal and assembler expressions

Expression elements

Expression elements: Overview

Constants

Registers

Symbols

Expression classes

Expression types

Expression operators

Assembler procedures and functions

Object Pascal grammar

Formal grammar